UNIVERSITY OF OSLO
Department of Informatics

A Probabilistic Strategy Language for Probabilistic Rewrite Theories and its Application to Cloud Computing

Research Report 431

Lucian Bentea
Peter Csaba Ölveczky

ISBN 82-7368-396-6
ISSN 0806-3036

June 19, 2014

A Probabilistic Strategy Language for Probabilistic Rewrite Theories and its Application to Cloud Computing

Lucian Bentea[1] and Peter Csaba Ölveczky[1,2]

[1] University of Oslo
[2] University of Illinois at Urbana-Champaign

Abstract. Several formal models combine probabilistic and nondeterministic features. To allow their probabilistic simulation and statistical model checking by means of pseudo-random number sampling, all sources of nondeterminism must first be quantified. However, current tools offer limited flexibility for the user to define how the nondeterminism should be quantified. In this report we propose an expressive *probabilistic strategy language* that allows the user to define complex strategies for quantifying the nondeterminism in *probabilistic rewrite theories*. These strategies may depend on the current system state, and their associated weight expressions can be given by any computable function defined equationally in Maude. We have implemented PSMaude, a tool that extends Maude with a probabilistic simulator and a statistical model checker for our language. We illustrate the convenience of being able to define different probabilistic strategies by a cloud computing example, where a (non-probabilistic) rewrite theory defines the capabilities of the cloud computing infrastructure, and where different load balancing policies are specified by different probabilistic strategies. Our language also enables a Maude-based safety/QoS modeling and analysis methodology in which key safety properties can be verified for a basic "uncluttered" non-probabilistic model, and where QoS properties for different probabilistic strategies can be analyzed by probabilistic simulation and statistical model checking.

1 Introduction

Many formal analysis tools support the modeling of systems that exhibit both probabilistic and nondeterministic behaviors. To allow their probabilistic simulation and statistical model checking using pseudo-random number sampling, the nondeterminism must be quantified to obtain a fully probabilistic model. However, there is typically limited support for user-definable adversaries to quantify the nondeterminism in reasonably expressive models; such adversaries are either added by the tool or must be encoded directly into the system model.

In this report we propose an expressive *probabilistic strategy language* for *probabilistic rewrite theories* [21, 1] that allows users to define complex adversaries for a model, and therefore allows us to separate the definition of the system model from that of the adversary needed to quantify the nondeterminism in the system model.

Rewriting logic is a simple and expressive logic for concurrent systems in which the data types are defined by an algebraic equational specification and where the local transition patterns are defined by conditional labeled rewrite rules of the form $l : t \longrightarrow t'$ **if** *cond*, where l is a label and t and t' are terms (typically with variables) representing state fragments. The Maude system [13] is a high-performance simulation, reachability, and LTL model checking tool for rewriting logic that has been successfully applied to many large state-of-the-art applications (see, e.g., [30, 26] for an overview).

Rewriting logic has been extended to *probabilistic rewrite theories* [21, 1], where probabilities are introduced by new variables in the righthand side term t' of a rewrite rule. These new variables are instantiated according to a probability distribution associated with the rewrite rule. Probabilistic rewrite theories, together with the VeStA statistical model checker [33], have been applied to analyze sensor network algorithms [20] and defense mechanisms against denial of service attacks [3, 15]. However, since the probabilistic rewrite theories were highly nondeterministic, adversaries had to be encoded into the model before any analysis could take place.

Probabilistic model checking suffers from state space explosion which renders it unfeasible for automated analysis of the complex concurrent systems targeted by rewriting logic. *Statistical model checking* [23, 34, 32] trades absolute confidence in the correctness of the model checking for computational efficiency, and essentially consists of simulating a number of different system behaviors until a certain confidence level is reached. This not only makes statistical analysis feasible, but also makes such model checking amenable to parallelization, which is exploited in the parallel version PVeStA [2] of the statistical model checker VeStA.

For any meaningful statistical/probabilistic reasoning to take place, the models must be *fully probabilistic*; that is, they should not exhibit any unquantified nondeterminism. However, probabilistic rewrite theories typically have both probabilistic and nondeterministic behaviors. Given two probabilistic rewrite rules

$$l_1 : \; f(X) \longrightarrow Y \;\; \textbf{with probability } Y := \ldots$$
$$l_2 : \; g(X) \longrightarrow Y \;\; \textbf{with probability } Y := \ldots$$

there is a nondeterministic choice concerning *which* rule to apply to a term $h(f(f(a)), g(a), g(b))$, and, once a rule is selected, we have a nondeterministic choice of *where* in the term the rule is applied (is l_1 applied to $f(f(a))$ or to $f(a)$? is l_2 applied to $g(a)$ or to $g(b)$?). For *flat* (i.e., non-hierarchical) *object-oriented* specifications, the paper [1] proposes to schedule any new event at a given future fictitious time, where the corresponding delay is set according to an (exponential) probability distribution over a dense time domain. This resolves the nondeterminism since the probability of scheduling two events at the same time is zero. However, this approach does not give the user much flexibility in resolving the nondeterminism and is restricted to a limited subclass of probabilistic rewrite theories. On the other hand, forcing the modeler to specify fully probabilistic systems would lead to ugly and cumbersome specifications (where, for example, all rewrite rules would have to be "global") and is highly undesirable.

In this report we propose and implement a *probabilistic strategy language* for probabilistic rewrite theories that can be used to define memoryless probabilistic strategies to quantify all sources of nondeterminism in any probabilistic rewrite theory. To analyze such theories under given probabilistic strategies, we have also formalized and integrated into (Full) Maude both probabilistic simulation and statistical model checking. (Describing this formalization is beyond the scope of this report.) Our strategy language and its implementation, the PSMaude tool [5], enable a Maude-based safety/QoS modeling and analysis methodology in which:

1. A non-probabilistic rewrite theory defines all possible behaviors in a simple "uncluttered" way; this model can then be directly subjected to important safety analyses to guarantee the absence of bad behaviors.
2. Different QoS policies and/or probabilistic environments can then be defined as probabilistic strategies on top of the basic verified model for QoS reasoning by probabilistic simulation and statistical model checking.

We exemplify in Section 6 the usefulness of this methodology and of the possibility to define different complex probabilistic strategies on top of the same model with a cloud computing example, where a (non-probabilistic) rewrite theory defines all the possible ways in which requested resources can be allocated on servers in the cloud, as well as all possible environment behaviors. We then use standard techniques to prove safety properties of this model. However, for QoS and other purposes, one could imagine a number of different *policies* for assigning resources to service providers and users, such as, e.g.,

- Service providers might request virtual machines uniformly across different regions (for fault-tolerance and omni-presence), or with higher probability at certain locations, or with higher probability at more stable servers.
- Service users may be assigned virtual machines either closer to their geographical locations with high probability, or on physical servers with low workload, or on reliable servers, with high probability.

Each load balancing policy can be naturally specified as a probabilistic strategy on top of the (non-probabilistic) model of the cloud computing infrastructure that has already been proved to be "safe." We then use PSMaude to perform probabilistic simulation and statistical model checking to analyze the QoS effect of the different load balancing policies.

The rest of the report is structured as follows. Section 2 presents some background on rewriting logic, probabilistic rewrite theories and statistical model checking. In Section 3 we define memoryless adversaries that resolve all nondeterminism in probabilistic rewrite theories, and show that the PCTL model checking problem of probabilistic rewrite theories under a memoryless adversary is well-defined. Our probabilistic strategy language is introduced in Section 4 which gives its syntax and semantics, and defines the class of "well-behaved" probabilistic strategies. Section 5 gives details about our tool and includes a simple system specification and its analysis in PSMaude. In Section 6 we show the usefulness of our methodology through a cloud computing example and its QoS analysis under different probabilistic load balancing policies by means of simulation and statistical model checking. Section 7 discusses related work and compares our approach to similar ones in the literature. Finally, Section 8 gives some concluding remarks and suggests topics for future work.

2 Preliminaries

Rewriting Logic and Maude. A *rewrite theory* [28] is a tuple $\mathcal{R} = (\Sigma, E \cup A, L, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic theory [29], with E a set of (possibly conditional) equations $(\forall \vec{x})\ \ t = t'\ \ \textbf{if}\ \ cond$ and membership axioms $(\forall \vec{x})\ \ t : s\ \ \textbf{if}\ \ cond$, where t and t' are Σ-terms, s is a sort, and $cond$ is a conjunction of equalities and sort memberships, and with A a collection of *structural axioms* specifying properties of operators, like commutativity, associativity, etc. Furthermore, R is a set of labeled conditional rewrite rules

$$(\forall \vec{x})\ \ l :\ t \longrightarrow t'\ \ \textbf{if}\ \ cond, \tag{1}$$

where $l \in L$ is a label, t and t' are terms of the same kind, $cond$ is a conjunction of equalities, memberships and rewrites, and $\vec{x} = vars(t) \cup vars(t') \cup vars(cond)$. Such a rule specifies a local transition from an instance of the term t to the corresponding instance of the term t', provided that the condition $cond$ is satisfied by the instantiating substitution. We write $vars(t)$ for the set of variables occurring in a term t; if $vars(t) = \emptyset$, then t is called a *ground term*. If E is terminating, confluent and sort-decreasing modulo A, then $Can_{\Sigma,E/A}$ denotes the algebra

of A-equivalence classes of fully simplified ground terms, or "normal forms," and we denote by $[t]_A \in Can_{\Sigma,E/A}$ the A-equivalence class of a fully simplified term t. An *E/A-canonical ground substitution* for a set of variables $\vec{x}$ is a function $[\theta]_A : \vec{x} \to Can_{\Sigma,E/A}$; we denote by $CanGSubst_{E/A}(\vec{x})$ the set of all such functions. We use the same notation $[\theta]_A$ for the homomorphic extension of $[\theta]_A$ to Σ-terms. A *context* is a Σ-term $\mathbb{C}$ with a single occurrence of a single variable, denoted $\odot$ and called the *hole*. Two contexts $\mathbb{C}$ and $\mathbb{C}'$ are A-equivalent if $A \vdash (\forall \odot)\ \mathbb{C}(\odot) = \mathbb{C}'(\odot)$. In what follows, for simplicity, we also call the A-equivalence class $[\mathbb{C}]_A$ a context.

Given $[u]_A \in Can_{\Sigma,E/A}$, its *$R/A$-matches* are triples $([\mathbb{C}]_A, r, [\theta]_A)$ where $[\mathbb{C}]_A \in Can_{\Sigma,E/A}(\odot)$ is a context, $r \in R$ is a rewrite rule of the form (1), $[\theta]_A \in CanGSubst_{E/A}(\vec{x})$ is a substitution such that $E \cup A \vdash \theta(cond)$, and $[u]_A = [\mathbb{C}(\odot \leftarrow \theta(t))]_A$. We denote by $\mathcal{M}([u]_A)$ the set of all R/A-matches of a term $[u]_A$; $[u]_A$ is a *deadlock* term if $\mathcal{M}([u]_A) = \emptyset$. Given terms $[u]_A, [v]_A \in Can_{\Sigma,E/A}$, an *$E/A$-canonical one-step rewrite* [21, 1] from $[u]_A$ to $[v]_A$ is a labeled transition $[u]_A \xrightarrow{([\mathbb{C}]_A, r, [\theta]_A)} [v]_A$, where $([\mathbb{C}]_A, r, [\theta]_A)$ is an R/A-match for $[u]_A$, and $[v]_A = [\mathbb{C}(\odot \leftarrow \theta(t'))]_A$ is the result of the one-step rewrite. We define the set of rules that are *enabled* for a term $[u]_A$, the set of *valid contexts* for $[u]_A$ and a rule r, and the set of *valid substitutions* for $[u]_A$, a rule r, and a context $[\mathbb{C}]_A$, respectively, in the expected way:

$$\begin{aligned}
\texttt{enabled}([u]_A) &= \{r \in R \mid \exists [\mathbb{C}]_A, \exists [\theta]_A : \ ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\} \\
C([u]_A, r) &= \{[\mathbb{C}]_A \in Can_{\Sigma,E/A}(\odot) \mid \exists [\theta]_A : \ ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\} \\
S([u]_A, r, [\mathbb{C}]_A) &= \{[\theta]_A \in CanGSubst_{E/A}(\vec{x}) \mid ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\}
\end{aligned}$$

Maude [13] is a high-performance simulation, reachability analysis, and LTL model checking tool for rewrite theories. We use Maude syntax, so that conditional rewrite rules are written `crl [`l`]:` t `=>` t' `if` *cond*. In object-oriented Maude specifications [13], the system state is a term of sort `Configuration` denoting a multiset of objects and messages, with multiset union denoted by juxtaposition. A class declaration `class` C `|` $att_1 : s_1$`, ...,` $att_n : s_n$ declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$, respectively. A *subclass* inherits the attributes and rules of its superclass(es). Objects are represented as terms of the form `<` $o : C \mid att_1 : val_1,\ \ldots,\ att_n : val_n$ `>`, where o is the object's identifier of sort `Oid`, C is the object's class, and where $val_1, \ldots, val_n$ are the current values of the object's attributes $att_1, \ldots, att_n$. For example, the rule

```
rl [l]: m(O, w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  m'(O', x) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of `O` and send a new message `m'(O',x)`. "Irrelevant" attributes (such as `a3` and the righthand side occurrence of `a2`) need not be mentioned.

Markov Chains. Given a set $\Omega \neq \emptyset$, a *σ-algebra* over Ω is a collection of sets $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ such that $\Omega \setminus F \in \mathcal{F}$ for all $F \in \mathcal{F}$, and $\bigcup_{i \in I} F_i \in \mathcal{F}$ for all collections $\{F_i\}_{i \in I} \subseteq \mathcal{F}$ indexed by a finite or countably infinite set I. Given a σ-algebra $\mathcal{F}$ over Ω, a function $\mathbb{P} : \mathcal{F} \to [0, 1]$ is called a *probability measure* if $\mathbb{P}(\Omega) = 1$ and $\mathbb{P}(\cup_{i \in I} F_i) = \sum_{i \in I} \mathbb{P}(F_i)$, for all collections $\{F_i\}_{i \in I} \subseteq \mathcal{F}$ of pairwise disjoint sets. The triple $(\Omega, \mathcal{F}, \mathbb{P})$ is then called a *probability space*. We denote by $PMeas(\Omega, \mathcal{F})$ the set of all probability measures on $\mathcal{F}$ over Ω. A function $p : \Omega \to [0, 1]$ with the property that $\sum_{\omega \in \Omega} p(\omega) = 1$ is called a *probability mass function* (pmf). If Ω is finite or countably infinite, a probability mass uniquely defines a probability

measure $\mathbb{P} : \mathcal{F} \to [0,1]$ via $\mathbb{P}(A) = \sum_{a \in A} p(a)$, for all sets $A \in \mathcal{F}$. Furthermore, a family of pmf's can be used to define the behavior of a (memoryless) probabilistic system. In particular, a *discrete-time Markov chain* (DTMC) is given by a countable set of *states* S, and a *transition probability* matrix $T : S \times S \to [0,1]$, where $T(s) : S \to [0,1]$ is a pmf for all states $s \in S$, i.e., $T(s, s')$ is the probability for the DTMC to make a transition from state s to state s'.

Probabilistic Rewrite Theories. In *probabilistic rewrite theories* (PRTs) [21] the righthand side t' of a rule $l : t \longrightarrow t'$ **if** *cond* may contain variables $\vec{y}$ that do not occur in t, and that are instantiated according to a probability measure taken from a *family* of probability measures—one for each instance of the variables in t—associated with the rule. Formally, a PRT $\mathcal{R}_\pi$ is a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a rewrite theory, and π maps each rule r of $\mathcal{R}$, with $vars(t) = \vec{x}$ and $vars(t') \setminus vars(t) = \vec{y}$, to a mapping

$$\pi_r : [\![cond(\vec{x})]\!] \to PMeas\left(CanGSubst_{E/A}(\vec{y}), \mathcal{F}_r\right),$$

where $[\![cond(\vec{x})]\!] = \{\, [\theta]_A \in CanGSubst_{E/A}(\vec{x}) \mid E \cup A \vdash \theta(cond) \,\}$, and $\mathcal{F}_r$ is a σ-algebra over $CanGSubst_{E/A}(\vec{y})$. That is, for each substitution $[\theta]_A$ of the variables in t that satisfies *cond*, we get a probability measure $\pi_r([\theta]_A)$ for instantiating the variables $\vec{y}$. The rule r together with π_r is called a *probabilistic rewrite rule*, and is written:

$$l : \; t \longrightarrow t' \;\; \textbf{if } cond \;\; \textbf{with probability } \pi_r$$

We refer to the specification of the "blackboard game" in Section 5 for an example of the syntax used to specify probabilistic rewrite rules and the probability measure π_r. An E/A-*canonical one-step rewrite* of $\mathcal{R}_\pi$ [21, 1] is a labeled transition $[u]_A \xrightarrow{([\mathbb{C}]_A,\, r,\, [\theta]_A, [\rho]_A)} [v]_A$ with $m \triangleq ([\mathbb{C}]_A,\, r,\, [\theta]_A)$ a R/A-match for $[u]_A$, $[\rho]_A \in CanGSubst_{E/AS}(\vec{y})$, and where $[v]_A = [\mathbb{C}(\odot \leftarrow (\theta \cup \rho)(t'))]_A$. To quantify the nondeterminism in the choice of m, the notion of *adversary* is introduced in [21, 1] that samples m from a pmf that depends on the computation history. A *memoryless* adversary[3] is a family of pmf's $\{\sigma_{[u]_A} : \mathcal{M}([u]_A) \to [0,1]\}_{[u]_A}$, where $\sigma_{[u]_A}(m)$ is the probability of picking the R/A-match m. A consequence of a result in [21] is that executing $\mathcal{R}_\pi$ under $\{\sigma_{[u]_A}\}_{[u]_A}$ is described by a DTMC.

PCTL. The *probabilistic computation tree logic* (PCTL) [18] extends CTL [12] with an operator $\mathcal{P}$ to express properties of DTMCs. We use a subset of PCTL, without time-bounded and steady-state operators. If AP is a set of atomic propositions, ϕ is a *state* formula, and ψ is a *path* formula, PCTL formulas over AP in this sublogic are defined by:

$$\phi ::= true \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}(\psi) \qquad \psi ::= \phi \,\mathcal{U}\, \phi \mid \mathbf{X}\phi$$

where $a \in AP$, $\bowtie \in \{<, \leq, >, \geq\}$, and $p \in [0,1]$. PCTL satisfaction is defined over DTMCs, e.g., the meaning of $\mathcal{M}, s \models \mathcal{P}_{<0.05}(true\ \mathcal{U}\ \phi)$ is that ϕ eventually becomes true in less than 5% of all the runs of the DTMC $\mathcal{M}$ from state s.

Statistical Model Checking and VeStA. Traditional model checking suffers from the state space explosion problem, whereas *statistical model checking* [23, 34, 32] trades complete confidence for efficiency, allowing the analysis of large-scale probabilistic systems. This technique is based on simulating the model, and on performing statistical hypothesis testing to control the generation of execution traces. The simulation is stopped when a given *level of confidence* is

[3] This is a slightly modified version of the definition of adversaries in [21, 1].

reached for answering the model checking problem. The VeStA tool [33] supports statistical model checking and quantitative analysis of executable specifications in which all nondeterminism is quantified probabilistically. In VeStA, system properties are given in PCTL (or its continuous-time extension CSL [4]), while quantitative analysis queries are given as *quantitative temporal expressions* in the QuaTEx logic [1]. Apart from being able to specify any PCTL or CSL formula, QuaTEx queries may ask for the expected values of quantities associated with the model—VeStA runs Monte Carlo simulations of the model and provides estimates for these expected values.

3 Adversaries for Probabilistic Rewrite Theories

In this section we introduce *memoryless adversaries* that allow quantifying all nondeterministic choices in a probabilistic rewrite theory. We also show that probabilistic rewrite theories controlled by memoryless adversaries are semantically equivalent to discrete-time Markov chains, possibly with infinite state spaces.

In an E/A-canonical one-step rewrite $[u]_A \xrightarrow{([\mathbb{C}]_A,\, r,\, [\theta]_A)} [v]_A$ the source of nondeterminism is the choice of the R/A-match $([\mathbb{C}]_A,\, r,\, [\theta]_A)$ for $[u]_A$. To quantify all nondeterminism, we must select the R/A-matches $([\mathbb{C}]_A,\, r,\, [\theta]_A)$ according to some probability distribution; the probabilistic strategy language proposed in this paper serves the purpose of specifying the probability distribution for selecting a R/A-match at each state $[u]_A$. We focus on *memoryless* adversaries, i.e., adversaries that only depend on the current state $[u]_A$.[4]

Definition 1. *A* memoryless adversary[5] *of a probabilistic rewrite theory $\mathcal{R}_\pi$ is a family of probability mass functions $\{\sigma_{[u]_A} : \mathcal{M}([u]_A) \to [0,1] \mid [u]_A \in Can_{\Sigma,E/A}\}$, where $\sigma_{[u]_A}([\mathbb{C}]_A, r, [\theta]_A)$ is the probability of selecting the R/A-match $([\mathbb{C}]_A, r, [\theta]_A)$ for the state $[u]_A$.*

The following result states that the semantics of executing a probabilistic rewrite theory under a memoryless adversary is given by a discrete-time Markov chain. This shows that probabilistic rewrite theories in which all nondeterminism is resolved by means of a memoryless adversary are amenable to statistical model checking.

Proposition 1. *The execution of a probabilistic rewrite theory under a memoryless adversary is described by a (possibly infinite state space) DTMC.*

Proof (Sketch). The DTMC (S, T) uniquely determined by a probabilistic rewrite theory $\mathcal{R} = (\Sigma, E \cup A, L, R)$ and a memoryless adversary $\{\sigma_{[u]_A}\}_{[u]_A}$, has set of states $S = Can_{\Sigma,E/A}$ and transition matrix $T : Can_{\Sigma,E/A} \times Can_{\Sigma,E/A} \to [0,1]$ defined by

$$T([u]_A, [v]_A) = \sum_{\substack{\alpha \in \mathcal{M}([u]_A) \\ [u]_A \xrightarrow{\alpha} [v]_A}} \sigma_{[u]_A}(\alpha)$$

for all $[v]_A \in Can_{\Sigma,E/A}$, provided that $[u]_A$ is not a deadlock term. If $[u]_A$ is a deadlock state, then we set $T([u]_A, [u]_A) = 1$; i.e., the DTMC (S, T) loops from deadlock states. □

[4] *History-dependent* adversaries are supported in our approach by combining memoryless adversaries with a slightly modified probabilistic rewrite theory, where each state is extended with information about the history of arriving in that state, i.e., the sequence of states, as well as the sequence of R/A-matches used to advance from one state in this sequence, to the next.

[5] This definition is a slightly modified version of the definition of adversaries in [21, 1].

Corollary 1. *Provided that the set of states reachable from the initial state is finite, the PCTL model checking problem of a probabilistic rewrite theory under a memoryless adversary is well-defined.*

Proof. Since DTMCs are particular classes of semi-Markov chains, and PCTL is a sublogic of CSL in which the time domain is discrete, the result follows from Proposition 1, and the fact that the CSL model checking problem is well-defined for semi-Markov chains [24]. □

The probability distribution associated with the choice of an R/A-match $([\mathbb{C}]_A, r, [\theta]_A)$ can be specified by means of the individual probability distributions corresponding to the choice of the rule r, the context $[\mathbb{C}]_A$, and the substitution $[\theta]_A$, which is usually more convenient than specifying their joint probability distribution. We therefore introduce notions of memoryless adversaries for rules, contexts, and substitutions, allowing us to more easily specify general memoryless adversaries, by means of the "product" of these individual adversaries.

Definition 2. *A* memoryless rule adversary *of $\mathcal{R}_\pi$ is a family of probability mass functions*

$$\left\{\mathcal{A}_{[u]_A} : \texttt{enabled}([u]_A) \to [0,1] \mid [u]_A \in \mathit{Can}_{\Sigma,E/A}\right\},$$

where $\mathcal{A}_{[u]_A}(r)$ is the probability of selecting the rule $r \in \texttt{enabled}([u]_A)$ for the state $[u]_A$.

Definition 3. *A* memoryless context adversary *of $\mathcal{R}_\pi$ is a family of probability mass functions*

$$\{\mathcal{A}^r_{[u]_A} : C([u]_A, r) \to [0,1] \mid [u]_A \in \mathit{Can}_{\Sigma,E/A},\ r \in \texttt{enabled}([u]_A)\},$$

such that $\mathcal{A}^r_{[u]_A}([\mathbb{C}]_A)$ is the conditional probability of selecting the context $[\mathbb{C}]_A \in C([u]_A, r)$, provided that the rule r was selected for the state $[u]_A$.

Definition 4. *A* memoryless substitution adversary *of $\mathcal{R}_\pi$ is a family of probability mass functions*

$$\{\mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A} : S([u]_A, r, [\mathbb{C}]_A) \to [0,1] \mid [u]_A \in \mathit{Can}_{\Sigma,E/A},\ r \in \texttt{enabled}([u]_A),\ [\mathbb{C}]_A \in C([u]_A, r)\}$$

where $\mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A}([\theta]_A)$ is the conditional probability of selecting the substitution $[\theta]_A \in S([u]_A, r, [\mathbb{C}]_A)$, provided that the rule r and the context $[\mathbb{C}]_A$ were selected for the state $[u]_A$.

By the chain rule of probability theory, i.e., the fact that any joint probability distribution can be defined by a product of conditional probabilities, we obtain the following "adversary decomposition" formula:

$$\sigma_{[u]_A}([\mathbb{C}]_A, r, [\theta]_A) = \mathcal{A}_{[u]_A}(r) \ \cdot\ \mathcal{A}^r_{[u]_A}([\mathbb{C}]_A) \ \cdot\ \mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A}([\theta]_A). \tag{2}$$

for all $([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)$, and all states $[u]_A \in \mathit{Can}_{\Sigma,E/A}$ of $\mathcal{R}_\pi$, In other words, by defining for each possible system state $[u]_A$, the *conditional* probability distributions $\mathcal{A}_{[u]_A}$, $\mathcal{A}^r_{[u]_A}$ and $\mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A}$ over the individual choices of rule, context, and substitution, we obtain a valid memoryless adversary $\{\sigma_{[u]_A}\}_{[u]_A}$. Furthermore, it is often more convenient to specify the probability distributions over the individual choices, than specifying the joint probability distribution associated with a memoryless adversary.

4 A Language for Specifying Probabilistic Strategies

The source of nondeterminism in a probabilistic rewrite theory is picking an R/A-match from a set of possible ones in each state. This section introduces a probabilistic strategy language that can be used to quantify this nondeterminism, i.e., for specifying memoryless adversaries of PRTs by means of probability distributions that may depend on the system state.

Remark 1. It is cumbersome to define *absolute* probabilities for each choice of rule, context and substitution (so that they add up to 1 in each state). If we have rules r_1, r_2, and r_3, and want r_1 to be applied with 3 times as high probability as r_2 (when both are enabled), which should be twice as likely as taking rule r_3, then, for a state $[u]_A$ where all rules are enabled, the probabilities would be $\{r_1 \mapsto 6/9, r_2 \mapsto 2/9, r_3 \mapsto 1/9\}$, and for a state $[u']_A$ where r_2 is not enabled, the distribution would be $\{r_1 \mapsto 6/7, r_3 \mapsto 1/7\}$, etc. This can soon become inconvenient. In our language one therefore instead defines *relative* weights for each rule, context, and substitution. That is, for any state $[u]_A$ in our example, the "weights" of the rules r_1, r_2 and r_3 could be 6, 2, and 1, respectively.

In what follows, we present the syntax and semantics of the PSMaude strategy language, and discuss what are the properties that a "well-behaved" probabilistic strategy expression should satisfy, i.e., one that can safely be given as input to PSMaude. The implementation of our tool PSMaude also provides an executable rewriting logic semantics for our probabilistic strategy language.

4.1 Syntax

The syntax for specifying memoryless rule, context and substitution adversaries of a probabilistic rewrite theory $\mathcal{R}_\pi$, is represented with the following abbreviated EBNF:

⟨*ProbStrat*⟩ ::= `psd` ⟨*Identifier*⟩ `:= <` ⟨*Identifier*⟩ `|` ⟨*Identifier*⟩ `|` ⟨*Identifier*⟩ `> .`

⟨*RuleStrat*⟩ ::= `psdrule` ⟨*Identifier*⟩ `:=`
`given state:` ⟨*StatePattern*⟩
`is:` ⟨*RulePMF*⟩ | `uniform`
[`if` ⟨*Condition*⟩] [`owise`] `.`

⟨*ContextStrat*⟩ ::= `psdcontext` ⟨*Identifier*⟩ `:=`
`given state:` ⟨*StatePattern*⟩
`rule:` ⟨*RulePattern*⟩
`is:` ⟨*ContextPMF*⟩ | `uniform`
[`if` ⟨*Condition*⟩] [`owise`] `.`

⟨*SubstStrat*⟩ ::= `psdsubst` ⟨*Identifier*⟩ `:=`
`given state:` ⟨*StatePattern*⟩
`rule:` ⟨*RulePattern*⟩
`context:` ⟨*ContextPattern*⟩
`is:` ⟨*SubstPMF*⟩ | `uniform`
[`if` ⟨*Condition*⟩] [`owise`] `.`

$\langle$*Condition*$\rangle$ specifies the condition under which the strategy can be applied, and `[owise]` specifies that it should be applied when no other strategy is applicable. A "joint" *probabilistic strategy expression* σ—specifying a memoryless adversary—is defined using the above syntax for $\langle$*ProbStrat*$\rangle$, while its constituent rule, context, and substitution strategy expressions—specifying memoryless rule, context and substitution adversaries—are defined using the syntax for $\langle$*RuleStrat*$\rangle$, $\langle$*ContextStrat*$\rangle$, and $\langle$*SubstStrat*$\rangle$, respectively. The remaining syntactic variables are given by:

$\langle$*StatePattern*$\rangle$ ::= a term in $Can_{\Sigma,E/A}(\vec{x})$ where $\vec{x}$ can be empty
$\langle$*RulePattern*$\rangle$ ::= $\langle$*RuleLabel*$\rangle$ | variable ranging over rule labels of $\mathcal{R}_\pi$
$\langle$*ContextPattern*$\rangle$ ::= a term in $Can_{\Sigma,E/A}(\{\odot\} \cup \vec{y})$ where $\vec{y} \subseteq \vec{x}$
$\langle$*SubstPattern*$\rangle$ ::= $\langle$*SubstConst*$\rangle$ | $\langle$*SubstVar*$\rangle$
$\langle$*SubstConst*$\rangle$ ::= substitution from the variables in the rules of $\mathcal{R}_\pi$ to $\vec{x}$
$\langle$*SubstVar*$\rangle$ ::= variable ranging over all substitutions of variables in $\mathcal{R}_\pi$

$\langle$*RulePMF*$\rangle$ ::= $\langle$*RulePattern*$\rangle \mapsto \langle$*Weight*$\rangle$ [; $\langle$*RulePMF*$\rangle$]
$\langle$*ContextPMF*$\rangle$ ::= $\langle$*ContextPattern*$\rangle \mapsto \langle$*Weight*$\rangle$ [; $\langle$*ContextPMF*$\rangle$]
$\langle$*SubstPMF*$\rangle$ ::= $\langle$*SubstPattern*$\rangle \mapsto \langle$*Weight*$\rangle$ [; $\langle$*SubstPMF*$\rangle$]
$\langle$*Weight*$\rangle$ ::= a term of sort `Rat` with variables $\vec{z} \subseteq \vec{x}$

4.2 Denotational Semantics

Let *RuleStratExp*, *ContextStratExp*, *SubstStratExp*, and *StratExp* be the syntactic categories of rule, context, and substitution strategy expressions,[6] as well as that of "joint" strategy expressions. The denotational semantics of our language is given by a semantic function:

$$\mathcal{D}_{\mathcal{R}_\pi} : \mathit{StratExp} \to \left(Can_{\Sigma,E/A} \to \mathit{pmf}\left(Can_{\Sigma,E/A}\right)\right)$$

that formalizes the probabilistic input/output behavior of a probabilistic rewrite theory $\mathcal{R}_\pi$ controlled by a probabilistic strategy expression, in one step of computation. Namely, $\mathcal{D}_{\mathcal{R}_\pi}[\![\sigma]\!]([u]_A)([v]_A)$ is the probability that $\mathcal{R}_\pi$ goes from state $[u]_A$ to state $[v]_A$ in one step under the strategy σ, i.e., $\mathcal{D}_{\mathcal{R}_\pi}[\![\sigma]\!]$ is the (possibly infinite-state) DTMC induced by the denotational semantics of executing $\mathcal{R}_\pi$ under the strategy σ.

In turn, $\mathcal{D}_{\mathcal{R}_\pi}$ is defined using the semantic functions $\mathcal{D}^R_{\mathcal{R}_\pi}$, $\mathcal{D}^C_{\mathcal{R}_\pi}$, and $\mathcal{D}^S_{\mathcal{R}_\pi}$ for rule, context, and substitution strategy expressions, respectively, as well as $\mathcal{D}^J_{\mathcal{R}_\pi}$, with signatures:

$$\begin{aligned}
\mathcal{D}^R_{\mathcal{R}_\pi} &: \mathit{RuleStratExp} \to \left(Can_{\Sigma,E/A} \to \overline{\mathit{pmf}}(R)\right)\\
\mathcal{D}^C_{\mathcal{R}_\pi} &: \mathit{ContextStratExp} \to \left(Can_{\Sigma,E/A} \times R \to \overline{\mathit{pmf}}\left(Can_{\Sigma,E/A}(\odot)\right)\right)\\
\mathcal{D}^S_{\mathcal{R}_\pi} &: \mathit{SubstStratExp} \to \Big(Can_{\Sigma,E/A} \times R \times Can_{\Sigma,E/A}(\odot) \to\\
&\qquad\qquad \to \overline{\mathit{pmf}}\left(\mathit{CanGSubst}_{E/A}(\vec{z})\right)\Big)\\
\mathcal{D}^J_{\mathcal{R}_\pi} &: \mathit{StratExp} \to \Big(Can_{\Sigma,E/A} \to\\
&\qquad\qquad \to \overline{\mathit{pmf}}\left(R \times Can_{\Sigma,E/A}(\odot) \times \mathit{CanGSubst}_{E/A}(\vec{z})\right)\Big)
\end{aligned}$$

[6] An expression is a *concatenation* of several, mutually exclusive subexpressions under the same identifier.

where $\vec{z} = \bigcup_{r \in R} vars(lhs(r))$ is the set of all variables in the lefthand sides of all rules of $\mathcal{R}_\pi$, $pmf(X) \stackrel{\Delta}{=} \{p : X \to [0,1] \mid \sum_{x \in X} p(x) = 1\}$ is the set of all pmf's on the set X, and $\overline{pmf}(X) \stackrel{\Delta}{=} pmf(X) \cup 0_X$, where the function $0_X : X \to \{0\}$ is identically zero on X, i.e., $0_X(x) = 0$ for all $x \in X$. They define:

- the probability $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}]\!]([u]_A)(r^*)$ that the rule strategy expression r assigns to selecting the rule r^* in state $[u]_A$;
- the probability $\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}]\!]([u]_A, r^*)([\mathbb{C}^*]_A)$ that the context strategy expression c assigns to selecting the context $[\mathbb{C}^*]_A$ in state $[u]_A$, after selecting rule r^*;
- the probability $\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A)$ that the substitution strategy expression s assigns to selecting the substitution $[\theta^*]_A$ in state $[u]_A$, after selecting rule r^* and context $[\mathbb{C}^*]_A$;
- the probability $\mathcal{D}^J_{\mathcal{R}_\pi}[\![\sigma]\!]([u]_A)(r^*, [\mathbb{C}^*]_A, [\theta^*]_A)$ that the joint strategy expression σ assigns to selecting the R/A-match $([\mathbb{C}^*]_A, r^*, [\theta^*]_A)$ in state $[u]_A$.

More precisely, the mathematical objects $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}]\!]$, $\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}]\!]$, and $\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}]\!]$ associated with any given rule, context, and substitution strategy expressions r, c, and s, are memoryless rule, context, and substitution adversaries, respectively. Furthermore, $\mathcal{D}^J_{\mathcal{R}_\pi}[\![\sigma]\!]$ is a "joint" memoryless adversary.

The function $\mathcal{D}_{\mathcal{R}_\pi}$ is then given by the following semantic equation:

$$\mathcal{D}_{\mathcal{R}_\pi}[\![\ \mathtt{psd}\ \langle \mathit{Identifier} \rangle\ \mathtt{:=}\ \mathtt{<}\ \mathit{RuleStratID}\ \mathtt{|}\ \mathit{ContextStratID}\ \mathtt{|}\ \mathit{SubstStratID}\ \mathtt{>}\ ]\!]([u]_A)([v]_A) =$$
$$\sum \Big[\mathcal{D}^J_{\mathcal{R}_\pi}[\![\ \mathtt{psd}\ \langle \mathit{Identifier} \rangle\ \mathtt{:=}\ \mathtt{<}\ \mathit{RuleStratID}\ \mathtt{|}\ \mathit{ContextStratID}\ \mathtt{|}\ \mathit{SubstStratID}\ \mathtt{>}\ ]\!]([u]_A)(r^*, [\mathbb{C}^*]_A, [\theta^*]_A) \cdot \pi_{r^*}([\theta^*]_A)([\rho^*]_A) \Big]$$

where the sum ranges over all E/A-canonical one-step rewrites $[u]_A \xrightarrow{([\mathbb{C}^*]_A, r^*, [\theta^*]_A, [\rho^*]_A)} [v]_A$, and where $\pi_{r^*}([\theta^*]_A)([\rho^*]_A)$ is the probability of selecting the substitution $[\rho^*]_A$ for the probabilistic variables of r^*, if r^* is a probabilistic rule, and is 1 otherwise. However, the above semantic equation holds in a state $[u]_A$ if and only if the sum in the righthand side is nonzero for at least one state $[v]_A$. Otherwise, we set:

$$\mathcal{D}_{\mathcal{R}_\pi}[\![\ \mathtt{psd}\ \langle \mathit{Identifier} \rangle\ \mathtt{:=}\ \mathtt{<}\ \mathtt{r}\ \mathtt{|}\ \mathtt{c}\ \mathtt{|}\ \mathtt{s}\ \mathtt{>}\ ]\!]([u]_A)([u]_A) = 1$$

i.e., we set loops from deadlock states $[u]_A$ with probability 1. Using the adversary decomposition formula (2), the function $\mathcal{D}^J_{\mathcal{R}_\pi}$ defines the semantics of a "joint" strategy expression by:

$$\mathcal{D}^J_{\mathcal{R}_\pi}[\![\ \mathtt{psd}\ \langle \mathit{Identifier} \rangle\ \mathtt{:=}\ \mathtt{<}\ \mathit{RuleStratID}\ \mathtt{|}\ \mathit{ContextStratID}\ \mathtt{|}\ \mathit{SubstStratID}\ \mathtt{>}\ ]\!]([u]_A)(r^*, [\mathbb{C}^*]_A, [\theta^*]_A)$$
$$= \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}]\!]([u]_A)(r^*) \cdot \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}]\!]([u]_A, r^*)([\mathbb{C}^*]_A) \cdot \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A)$$

where r, c, and s are the sets of all rule, context, and substitution strategy expressions with identifiers *RuleStratID*, *ContextStratID*, and *SubstStratID*, respectively.

Finally, we give the semantic equations for each of the functions $\mathcal{D}^R_{\mathcal{R}_\pi}$, $\mathcal{D}^C_{\mathcal{R}_\pi}$, and $\mathcal{D}^S_{\mathcal{R}_\pi}$. The main idea is to instantiate the weight expressions in each of the possible strategy expressions,

using the solution $[\theta]_A$ to the unification problem associated with the strategy expression and the current state $[u]_A$.

In what follows, we denote by $Sol_{cond}\left(t_1 \stackrel{?}{=} t'_1, \ldots, t_N \stackrel{?}{=} t'_N\right)$ the set of solutions $[\theta]_A$ to the given unification problem, i.e., such that $[\theta(t_i)]_A = [\theta(t'_i)]_A$ for all $i \in \{1, \ldots, N\}$, and that satisfy the condition $cond$, i.e., $E \cup A \vdash \theta(cond)$. We also use the Iverson bracket notation $[P] \stackrel{\Delta}{=} \texttt{if } P \texttt{ then } 1 \texttt{ else } 0 \texttt{ fi}$, for any logical statement P. Furthermore, for all ground terms t of sort `Rat`, the denotation $[\![t]\!] \in \mathbb{Q}$ is the unique rational number associated with t. We also denote by $r[l] \in R$ the unique rule in R whose label is l. (We assume that all rules in R are labeled, and *no two rules have the same label.*) Below $t(\vec{x})$ is a state pattern, i.e., a term with variables, $[\theta]_A \in CanGSubst_{E/A}(vars(t))$ is a substitution for t, $cond(\vec{x})$ is a term of sort `Bool` representing a Boolean condition, whereas $\lambda_1, \ldots, \lambda_N$ are rule labels, λ is either a variable over rule labels or a rule label, and $w_1(\vec{x}), \ldots, w_N(\vec{x})$ are terms of sort `Rat`, denoting their corresponding weights.

The functions $\mathcal{D}^R_{\mathcal{R}_\pi}$, $\mathcal{D}^C_{\mathcal{R}_\pi}$, and $\mathcal{D}^S_{\mathcal{R}_\pi}$ are defined by:

$$\mathcal{D}^R_{\mathcal{R}_\pi}[\![\,\texttt{psdrule } \langle Identifier\rangle := \texttt{given state: } t(\vec{x}) \texttt{ is: uniform if } cond(\vec{x}) \texttt{ .}\,]\!]([u]_A)(r^*)$$
$$= \left[Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right) \neq \emptyset\right] \cdot \frac{[r^* \in enabled([u]_A)]}{\#enabled([u]_A)}$$

$$\mathcal{D}^R_{\mathcal{R}_\pi}[\![\,\texttt{psdrule } \langle Identifier\rangle := \texttt{given state: } t(\vec{x}) \texttt{ is: } \lambda_1 \mapsto w_1(\vec{x}) \texttt{ ; } \ldots \texttt{ ; } \lambda_N \mapsto w_N(\vec{x})$$
$$\texttt{if } cond(\vec{x}) \texttt{ .}\,]\!]([u]_A)(r[\lambda_i])$$
$$= \left[\exists \theta \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right)\right] \cdot \left[r[\lambda_i] \in enabled([u]_A)\right] \cdot \frac{[\![\theta(w_i(\vec{x}))]\!]}{\sum\limits_{\substack{j=1 \\ r[\lambda_j] \in enabled([u]_A)}}^{N} [\![\theta(w_j(\vec{x}))]\!]}$$

$$\mathcal{D}^C_{\mathcal{R}_\pi}[\![\,\texttt{psdcontext } \langle Identifier\rangle := \texttt{given state: } t(\vec{x}) \texttt{ rule: } \lambda \texttt{ is: uniform}$$
$$\texttt{if } cond(\vec{x}) \texttt{ .}\,]\!]([u]_A, r^*)([\mathbb{C}^*]_A)$$
$$= \left[Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right) \neq \emptyset\right] \cdot \left[\lambda \stackrel{?}{=} r^*\right] \cdot \frac{\left[[\mathbb{C}]^*_A \in C([u]_A, r^*)\right]}{\#C([u]_A, r^*)}$$

$$\mathcal{D}^C_{\mathcal{R}_\pi}[\![\,\texttt{psdcontext } \langle Identifier\rangle := \texttt{given state: } t(\vec{x}) \texttt{ rule: } \lambda$$
$$\texttt{is: } c_1(\odot, \vec{x}) \mapsto w_1(\vec{x}) \texttt{ ; } \ldots \texttt{ ; } c_N(\odot, \vec{x}) \mapsto w_N(\vec{x}) \texttt{ if } cond(\vec{x}) \texttt{ .}\,]\!]([u]_A, r^*)([\mathbb{C}^*]_A)$$
$$= \left[\exists i \in \{1, \ldots, N\} \text{ s.t. } \exists \theta_{\mathbb{C}^*} \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\, c_i(\odot, \vec{x}) \stackrel{?}{=} \mathbb{C}^*\right)\right] \cdot \left[\lambda \stackrel{?}{=} r^*\right] \cdot$$
$$\cdot \frac{[\![\theta_{\mathbb{C}^*}(w_i(\vec{x}))]\!]}{\sum\limits_{\substack{[\widehat{\mathbb{C}}]_A \in C([u]_A, r^*) \\ \left[\exists j \in \{1, \ldots, N\} \text{ s.t. } \exists \theta_{\widehat{\mathbb{C}}} \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\, c_j(\odot, \vec{x}) \stackrel{?}{=} \widehat{\mathbb{C}}\right)\right]}} [\![\theta_{\widehat{\mathbb{C}}}(w_j(\vec{x}))]\!]}$$

$$\mathcal{D}^S_{\mathcal{R}_\pi}[\![\,\texttt{psdsubst}\ \langle \mathit{Identifier}\rangle := \texttt{given state:}\ t(\vec{x})\ \texttt{rule:}\ \lambda\ \texttt{context:}\ c(\odot,\vec{x})$$
$$\texttt{is: uniform if}\ \mathit{cond}(\vec{x})\ .]\!]\,([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*])$$
$$= \Big[Sol_{\mathit{cond}(\vec{x})}\Big(t(\vec{x}) \stackrel{?}{=} u,\, c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}^*\Big) \neq \emptyset\Big] \cdot \Big[\lambda \stackrel{?}{=} r^*\Big] \cdot \frac{\Big[[\theta]^*_A \in S([u]_A, r^*, [\mathbb{C}^*]_A)\Big]}{\#S([u]_A, r^*, [\mathbb{C}^*]_A)}$$

$$\mathcal{D}^S_{\mathcal{R}_\pi}[\![\,\texttt{psdsubst}\ \langle \mathit{Identifier}\rangle := \texttt{given state:}\ t(\vec{x})\ \texttt{rule:}\ \lambda\ \texttt{context:}\ c(\odot,\vec{x})$$
$$\texttt{is:}\ s_1(\vec{z}\mapsto\vec{x})\mapsto w_1(\vec{x})\ \texttt{;}\ \ldots\ \texttt{;}\ s_N(\vec{z}\mapsto\vec{x})\mapsto w_N(\vec{x})\ \texttt{if}\ \mathit{cond}(\vec{x})\ .]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A)$$
$$= \Big[\exists i \in \{1,\ldots,N\}, \exists \Omega_{\theta^*} \in Sol_{\mathit{cond}(\vec{x})}\Big(t(\vec{x}) \stackrel{?}{=} u, c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}^*\Big)\colon [s_i(\vec{z}\mapsto\vec{x})\circ\Omega_{\theta^*}]_A = [\theta^*]_A\Big]$$
$$\cdot \Big[\lambda \stackrel{?}{=} r^*\Big] \cdot \frac{[\![\Omega_{\theta^*}(w_i(\vec{x}))]\!]}{\displaystyle\sum_{\substack{[\widehat{\theta}]_A \in S([u]_A, r^*, [\mathbb{C}^*]_A)\\ \exists j\in\{1,\ldots,N\}, \exists \Omega_{\widehat{\theta}} \in Sol_{\mathit{cond}(\vec{x})}\\ \left(t(\vec{x}) \stackrel{?}{=} u,\, c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}^*\right)\ \text{s.t.}\ \left[s_j(\vec{z}\mapsto\vec{x})\circ\Omega_{\widehat{\theta}}\right]_A = [\widehat{\theta}]_A}} [\![\Omega_{\widehat{\theta}}(w_j(\vec{x}))]\!]}$$

The semantics of "composite" strategy expressions is obtained recursively from the semantics of their constituents:

$$\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_1\ \mathsf{r}_2]\!] = \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_1]\!] + \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_2]\!]$$
$$\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_1\ \mathsf{c}_2]\!] = \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_1]\!] + \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_2]\!]$$
$$\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_1\ \mathsf{s}_2]\!] = \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_1]\!] + \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_2]\!]$$

where $(f+g)(x) \stackrel{\Delta}{=} f(x) + g(x)$ for all functions $f, g : X \to Y$ and all $x \in X$. Finally, the semantics of the optional **[owise]** statement is given by the following equations:

$$\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_{\mathbf{nonowise}} \uplus \mathsf{r}_{\mathbf{owise}}]\!]([u]_A)(r^*) =$$
$$= \begin{cases} \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_{\mathbf{nonowise}}]\!]([u]_A)(r^*), & \text{if } \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_{\mathbf{nonowise}}]\!]([u]_A)(r^*) \neq 0 \\ \mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}_{\mathbf{owise}}]\!]([u]_A)(r^*), & \text{otherwise} \end{cases}$$

$$\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_{\mathbf{nonowise}} \uplus \mathsf{c}_{\mathbf{owise}}]\!]([u]_A, r^*)([\mathbb{C}^*]_A) =$$
$$= \begin{cases} \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_{\mathbf{nonowise}}]\!]([u]_A, r^*)([\mathbb{C}^*]_A), & \text{if } \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_{\mathbf{nonowise}}]\!]([u]_A, r^*)([\mathbb{C}^*]_A) \neq 0 \\ \mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}_{\mathbf{owise}}]\!]([u]_A, r^*)([\mathbb{C}^*]_A), & \text{otherwise} \end{cases}$$

$$\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_{\mathbf{nonowise}} \uplus \mathsf{s}_{\mathbf{owise}}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A) =$$
$$= \begin{cases} \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_{\mathbf{nonowise}}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A), & \text{if } \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_{\mathbf{nonowise}}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A) \neq 0 \\ \mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}_{\mathbf{owise}}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A), & \text{otherwise} \end{cases}$$

where $\mathsf{r}_{\mathbf{owise}}$ and $\mathsf{r}_{\mathbf{nonowise}}$ are the disjoint sets of rule strategy expressions that are annotated using the **[owise]** statement, and those that are not, respectively. The sets $\mathsf{c}_{\mathbf{owise}}$, $\mathsf{c}_{\mathbf{nonowise}}$, $\mathsf{s}_{\mathbf{owise}}$, and $\mathsf{s}_{\mathbf{nonowise}}$ are defined simiarly.

To avoid division by zero in the above semantic equations, we also make the convention that, if any of the sets $\mathtt{enabled}([u]_A)$, $C([u]_A, r^*)$, or $S([u]_A, r^*, [\mathbb{C}^*]_A)$ is empty, then we set the value of the corresponding denotation to zero. Furthermore, from the above equations, it follows by direct calculation that:

$$\sum_{[v]_A \in Can_{\Sigma,E/A}} \mathcal{D}_{\mathcal{R}_\pi}[\![\sigma]\!]([u]_A)([v]_A) = 1$$

for all $[u]_A$ and all strategy expressions σ. Therefore, $\mathcal{D}_{\mathcal{R}_\pi}[\![\sigma]\!]([u]_A)$ indeed defines a pmf in all states $[u]_A$.

Notice that in the above equations the following sets need to be computed:

- The set $\mathtt{enabled}([u]_A)$ of all enabled rules in $[u]_A$;
- The set $C([u]_A, r^*)$ of all valid contexts that match $[u]_A$ with the lefthand side of r^*;
- The set $S([u]_A, r^*, [\mathbb{C}^*]_A)$ of all valid substitutions that can be used to match $[u]_A$ with the lefthand side of r^* and using the context $[\mathbb{C}^*]_A$.

This can be computationally intensive, e.g., the size of $C([u]_A, r^*)$ can be exponential in the size of the state $[u]_A$, due to commutative operators in $[u]_A$.

4.3 Operational Semantics

A "joint" probabilistic strategy expression σ declared as

```
psd StratID := < RuleStratID | ContextStratID | SubstStratID >
```

quantifies all nondeterminism in a probabilistic rewrite theory $\mathcal{R}_\pi$ given as a probabilistic module, as follows. Let r, c, and s be the sets of all rule, context, and substitution strategy expressions with identifiers *RuleStratID*, *ContextStratID*, and *SubstStratID*, respectively. Given the current non-deadlock state $[u]_A$, one step of computation from $[u]_A$ of $\mathcal{R}_\pi$ under the memoryless adversary $\mathcal{D}^J_{\mathcal{R}_\pi}[\![\sigma]\!]$ defined by σ is described as:

1. The pmf $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}]\!]([u]_A)$ is computed, from which a rule r^* is sampled.
2. The pmf $\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathsf{c}]\!]([u]_A, r^*)$ is computed, from which a context $[\mathbb{C}^*]_A$ is sampled.
3. The pmf $\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathsf{s}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)$ is computed, from which a substitution $[\theta^*]_A$ is sampled.
4. If r^* is a probabilistic rewrite rule, a substitution $[\rho^*]_A$ for its probabilistic variables $\vec{y}$ is sampled from the distribution $\pi_{r^*}([\theta^*]_A)$; otherwise, we set $\rho^* = \emptyset$.
5. The new state is computed as $[v]_A = [\mathbb{C}^*(\odot \leftarrow (\theta^* \cup \rho^*)(rhs(r^*)))]_A$.

Remark 2. This semantics extends the operational semantics of probabilistic rewriting in Definition 5 of [1], to probabilistic rewriting under a given probabilistic strategy.

4.4 Well-Behaved Probabilistic Strategies

In this section we introduce "well-behaved" probabilistic strategies, a property that can be checked to ensure that successor states can be sampled in each (non-deadlock) system state under a given strategy. An example of a probabilistic strategy that is not well-behaved is given at the end of this section.

We provide a series of definitions to make precise what it means for a probabilistic strategy to be "well-behaved". To produce correct system trajectories, the PSMaude tool implementing our strategy language assumes, but does not check, that the given strategy is well-behaved; it is up to the user to ensure that this is indeed the case.

Strategies in canonical form. For efficiency reasons, our tool assumes that the given probabilistic strategy is in "canonical" form, in the following sense:

Definition 5. *We say that a probabilistic strategy σ is in* canonical form *if all the rule, context and substitution patterns (which are associated some weight expressions) are (syntactically) distinct from one another as terms with variables.*

Remark 3. Any probabilistic strategy σ can be brought to canonical form by summing up the weight expressions for identical rule/context/substitution patterns, and assigning them to a single pattern.

Blocking strategies. It may happen that a probabilistic strategy leads the probabilistic rewrite theory that it controls into additional deadlock states, which is usually not desired; we call such strategies "blocking" in the sense formalized below.

Definition 6. *We say that a rule strategy expression* $\mathtt{r}$ promotes *a rule* $r^* \in \mathtt{enabled}([u]_A)$ *in a state* $[u]_A$ *if* $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathtt{r}]\!]([u]_A)(r^*) > 0$*; otherwise we say that* $\mathtt{r}$ inhibits *the rule* r^* *in* $[u]_A$*. If there exists a state* $[u]_A \in Can_{\Sigma,E/A}$ *in which* $\mathtt{enabled}([u]_A) \neq \emptyset$ *but* $\mathtt{r}$ *inhibits* all *enabled rules, i.e.,* $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathtt{r}]\!]([u]_A)$ *is identically* 0 *on* $\mathtt{enabled}([u]_A)$*, then we say that* $\mathtt{r}$ *is* blocking*.*

Remark 4. We use the term "blocking" since a rule strategy expression $\mathtt{r}$ inhibiting all rules enabled in a non-deadlock state $[u]_A$ forcefully leads the probabilistic rewrite theory $\mathcal{R}_\pi$ into a deadlock state, which may be undesired.

We define similar concepts for context and substitution strategy expressions.

Definition 7. *We say that a context strategy expression* $\mathtt{c}$ promotes *a valid context* $[\mathbb{C}^*]_A \in C([u]_A, r^*)$ *for a state* $[u]_A$ *and a rule* $r^* \in \mathtt{enabled}([u]_A)$ *if* $\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathtt{c}]\!]([u]_A, r^*)([\mathbb{C}^*]_A) > 0$*; otherwise we say that* $\mathtt{c}$ inhibits *the context* $[\mathbb{C}^*]_A$ *in* $[u]_A$ *for rule* r^**. If there exists a state* $[u]_A \in Can_{\Sigma,E/A}$ *and a rule* $r^* \in \mathtt{enabled}([u]_A)$ *for which* $C([u]_A, r^*) \neq \emptyset$ *but* $\mathtt{c}$ *inhibits* all *valid contexts, i.e.,* $\mathcal{D}^C_{\mathcal{R}_\pi}[\![\mathtt{c}]\!]([u]_A, r^*)$ *is identically* 0 *on* $C([u]_A, r^*)$*, then we say that* $\mathtt{c}$ *is* blocking*.*

Definition 8. *We say that a substitution strategy expression* $\mathtt{s}$ promotes *a valid substitution* $[\theta^*]_A \in S([u]_A, r^*, [\mathbb{C}^*]_A)$ *for a state* $[u]_A$*, a rule* $r^* \in \mathtt{enabled}([u]_A)$ *and a context* $[\mathbb{C}^*]_A \in C([u]_A, r^*)$ *if* $\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathtt{s}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)([\theta^*]_A) > 0$*; otherwise we say that* $\mathtt{s}$ inhibits *the substitution* $[\theta^*]_A$ *in* $[u]_A$ *for rule* r^* *and context* $[\mathbb{C}^*]_A$*. If there exists a state* $[u]_A \in Can_{\Sigma,E/A}$*, a rule* $r^* \in \mathtt{enabled}([u]_A)$ *and a context* $[\mathbb{C}^*]_A \in C([u]_A, r^*)$ *for which* $S([u]_A, r^*, [\mathbb{C}^*]_A) \neq \emptyset$ *but* $\mathtt{s}$ *inhibits* all *valid substitutions, i.e.,* $\mathcal{D}^S_{\mathcal{R}_\pi}[\![\mathtt{s}]\!]([u]_A, r^*, [\mathbb{C}^*]_A)$ *is identically* 0 *on* $S([u]_A, r^*, [\mathbb{C}^*]_A)$*, then we say that* $\mathtt{s}$ *is* blocking*.*

Well-defined strategies. To be able to simulate a probabilistic rewrite theory under a given probabilistic strategy, the strategy must also be "well-defined", in the sense that we make precise below. Intuitively, the strategy must define a *single-valued* (as opposed to multi-valued) probability distribution, from which successor states can be sampled.

Definition 9. *A probabilistic rule strategy expression* $\mathtt{r}$ *in canonical form*

$$\begin{aligned} &\mathtt{psdrule}\ \langle \mathit{Identifier} \rangle := \mathtt{given\ state:}\ t(\vec{x}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{is:}\ \lambda_1 \mapsto w_1(\vec{x})\ \mathtt{;}\ \ldots\ \mathtt{;}\ \lambda_N \mapsto w_N(\vec{x}) \\ &\mathtt{if}\ \mathit{cond}(\vec{x})\ . \end{aligned} \tag{3}$$

is well-defined *if, for all states* $[u]_A \in Can_{\Sigma,E/A}$ *and all rules* $r[\lambda_i] \in \texttt{enabled}([u]_A)$*, the probability value*

$$\frac{[\![\theta(w_i(\vec{x})]\!]}{\sum\limits_{\substack{j=1 \\ r[\lambda_j]\in enabled([u]_A)}}^{N} [\![\theta(w_j(\vec{x}))]\!]}$$

is invariant under the change of solution $\theta \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right)$.

The following result, which we state without proof, gives a quick way to identify well-defined probabilistic rule strategy expressions. The general problem of checking whether a rule strategy is well-defined requires exploring the entire state space of a probabilistic rewrite theory $\mathcal{R}_\pi$ (and solving a unification problem in each state), and is therefore quite complex.

Proposition 2. *Let* r *be a probabilistic rule strategy expression of the form* (3)*. If the solution set* $Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right)$ *is a singleton for all states* $[u]_A \in Can_{\Sigma,E/A}$ *of* $\mathcal{R}_\pi$*, then* r *is well-defined.* □

Remark 5. The converse is not necessarily true since there exist rule strategy expressions r that are well-defined, and yet for some states $[u]_A$ the set $Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right)$ has at least two elements. However, rule strategy expressions for which $\left|Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u\right)\right| > 1$ for some state $[u]_A$ can easily become ill-defined if care is not taken when specifying the associated weight expressions $w_1(\vec{x}), \ldots, w_N(\vec{x})$. If at least one weight expression $w_i(\vec{x})$ has different values when instantiating it with two solutions $\theta \neq \theta'$, then the probabilistic rule strategy becomes ill-defined, and cannot be used to quantify the nondeterminism in $\mathcal{R}_\pi$.

Definition 10. *A "composite" probabilistic rule strategy expression* r *is well-defined if and only if all its constituents are well-defined.*

The above terminology stems from the fact that the semantic function $\mathcal{D}^R_{\mathcal{R}_\pi}$ is "well-defined" if and only if $\mathcal{D}^R_{\mathcal{R}_\pi}[\![\mathsf{r}]\!]([u]_A) : R \to [0,1]$ is a proper (single-valued) function, and not a multi-valued one. In what follows, we define similar concepts for probabilistic context strategy expressions.

Definition 11. *A probabilistic context strategy expression* c *in canonical form*

$$\begin{aligned} &\texttt{psdcontext}\ \langle \textit{Identifier}\rangle := \texttt{given state:}\ t(\vec{x}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{rule:}\ \lambda \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{is:}\ c_1(\odot,\vec{x}) \mapsto w_1(\vec{x})\ \texttt{;}\ \ldots\ \texttt{;}\ c_N(\odot,\vec{x}) \mapsto w_N(\vec{x}) \\ &\texttt{if}\ cond(\vec{x})\ . \end{aligned} \tag{4}$$

is well-defined *if, for all* $i \in \{1,\ldots,N\}$*, for all states* $[u]_A \in Can_{\Sigma,E/A}$ *of* $\mathcal{R}_\pi$*, for all rules* $r \in \texttt{enabled}([u]_A)$ *such that* $\lambda \stackrel{?}{=} r$ *has a solution, and for all contexts* $[\mathbb{C}]_A \in C([u]_A, r)$*, the probability value*

$$\frac{[\![\theta_{\mathbb{C}}(w_i(\vec{x}))]\!]}{\sum\limits_{\substack{[\widehat{\mathbb{C}}]_A \in C([u]_A,r) \\ \left[\exists j \in \{1,\ldots,N\}\ s.t.\ \exists \theta_{\widehat{\mathbb{C}}} \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\, c_j(\odot,\vec{x}) \stackrel{?}{=} \widehat{\mathbb{C}}\right)\right]}} [\![\theta_{\widehat{\mathbb{C}}}(w_j(\vec{x}))]\!]}$$

is invariant under the change of solution $\theta_{\mathbb{C}} \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\ c_i(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right)$.

We state without proof a similar result as for rule strategies, providing a quick way to identify well-defined probabilistic context strategy expressions:

Proposition 3. *Let* `c` *be a probabilistic context strategy expression of the form* (4)*. If the solution set* $Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\ c_i(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right)$ *is a singleton for all states* $[u]_A \in Can_{\Sigma,E/A}$ *of* $\mathcal{R}_\pi$*, all rules* $r \in \texttt{enabled}([u]_A)$ *such that* $\lambda \stackrel{?}{=} r$ *has a solution, all contexts* $[\mathbb{C}]_A \in C([u]_A, r)$*, and all* $i \in \{1,\ldots,N\}$*, then* `c` *is well-defined.* □

Remark 6. Again, the converse is not true, and a simple example of a well-defined probabilistic context strategy for which the above mentioned solution set has two elements, is given by the `BLACKBOARD` example in Section 5. However, probabilistic context strategy expressions for which $\#Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\ c_i(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right) > 1$ for some state $[u]_A$, some context $[\mathbb{C}]_A \in C([u]_A, r)$ and some $i \in \{1,\ldots,N\}$, can be ill-defined if the corresponding weight expressions $w_1(\vec{x}),\ldots,w_N(\vec{x})$ change their value when instantiated with different solutions θ from this set.

Definition 12. *A "composite" probabilistic context strategy expression* `c` *is well-defined if and only if all its constituents are well-defined.*

We define similar concepts for probabilistic substitution strategy expressions.

Definition 13. *A probabilistic substitution strategy expression* `s` *in canonical form*

$$\begin{array}{l}\texttt{psdsubst}\ \langle \textit{Identifier}\rangle := \texttt{given state:}\ t(\vec{x}) \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{rule:}\ \lambda \\ \qquad\qquad\qquad\qquad\quad \texttt{context:}\ c(\odot,\vec{x}) \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{is:}\ s_1(\vec{z}\mapsto\vec{x}) \mapsto w_1(\vec{x})\ \texttt{;}\ \ldots\ \texttt{;}\ s_N(\vec{z}\mapsto\vec{x}) \mapsto w_N(\vec{x}) \\ \texttt{if}\ cond(\vec{x})\ .\end{array} \tag{5}$$

is well-defined *if, for all* $i \in \{1,\ldots,N\}$*, for all states* $[u]_A \in Can_{\Sigma,E/A}$ *of* $\mathcal{R}_\pi$*, for all rules* $r \in \texttt{enabled}([u]_A)$ *such that* $\lambda \stackrel{?}{=} r$ *has a solution, for all contexts* $[\mathbb{C}]_A \in C([u]_A, r)$ *such that* $c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}$ *has a solution, and all substitutions* $[\theta]_A \in S([u]_A, r, [\mathbb{C}]_A)$*, the probability value*

$$\frac{[\![\Omega_\theta(w_i(\vec{x}))]\!]}{\displaystyle\sum_{\substack{[\widehat{\theta}]_A \in S([u]_A, r, [\mathbb{C}]_A) \\ \exists j \in \{1,\ldots,N\}, \exists \Omega_{\widehat{\theta}} \in Sol_{cond(\vec{x})} \\ \left(t(\vec{x}) \stackrel{?}{=} u,\ c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right)\ s.t.\ \left[s_j(\vec{z}\mapsto\vec{x}) \circ \Omega_{\widehat{\theta}}\right]_A = [\widehat{\theta}]_A}} [\![\Omega_{\widehat{\theta}}(w_j(\vec{x}))]\!]}$$

is invariant under the change of solution $\Omega_\theta \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\ c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right)$ *satisfying* $[s_i(\vec{z}\mapsto\vec{x}) \circ \Omega_\theta]_A = [\theta]_A$.

Proposition 4. *Let* `s` *be a probabilistic substitution strategy expression of the form* (5)*. If the set of solutions* $\Omega_\theta \in Sol_{cond(\vec{x})}\left(t(\vec{x}) \stackrel{?}{=} u,\ c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}\right)$ *satisfying* $[s_i(\vec{z}\mapsto\vec{x}) \circ \Omega_\theta]_A = [\theta]_A$ *is a singleton for all states* $[u]_A \in Can_{\Sigma,E/A}$ *of* $\mathcal{R}_\pi$*, all rules* $r \in \texttt{enabled}([u]_A)$ *such that* $\lambda \stackrel{?}{=} r$ *has a solution, all contexts* $[\mathbb{C}]_A \in C([u]_A, r)$ *such that* $c(\odot,\vec{x}) \stackrel{?}{=} \mathbb{C}$ *has a solution, all substitutions* $[\theta]_A \in S([u]_A, r, [\mathbb{C}]_A)$*, and all* $i \in \{1,\ldots,N\}$*, then* `s` *is well-defined.* □

Remark 7. The converse of this proposition is not true, e.g., probabilistic substitution strategy expressions can also be well-defined when their weight expressions are all symmetric functions of their arguments—so that the probability value in Definition 13 is invariant under a change of solution—even though the set mentioned in the above proposition is not a singleton.

Remark 8. As shown in the `BLACKBOARD` example in Section 5, an ill-defined probabilistic substitution strategy expression can easily be made well-defined by making it *conditional*. Unconditional substitution strategies combined with commutative operators may lead to such strategies being ill-defined, as illustrated in the example at the end of this section.

Definition 14. *A "composite" probabilistic substitution strategy expression* `s` *is well-defined if and only if all its constituents are well-defined.*

Finally, we formalize the notion of "well-defined" probabilistic strategy expressions:

Definition 15. *We say that a probabilistic strategy expression σ is* well-defined *if its underlying (composite) rule, context and substitution strategies are well-defined.*

Example of ill-defined strategies. In what follows we give simple, concrete examples of ill-defined probabilistic rule, context and substitution strategy expressions controlling a simple (non-probabilistic) rewrite theory. In this example it is rather obvious that, due to the asymmetry of some of the weight expressions, the corresponding strategy expressions are not well-defined. However, this becomes less obvious in strategies with more complex weight expressions.

Let $\mathcal{R} = (\Sigma, E \cup A, L, R)$ be a rewrite theory with the signature Σ having sorts `S` and `Nat`, a binary operator $f \in \Sigma_{\mathtt{S}\times\mathtt{S}\to\mathtt{S}}$, a *commutative* operator $g \in \Sigma_{\mathtt{Nat}\times\mathtt{Nat}\to\mathtt{S}}$ (so that $[\mathtt{g}(u,v)]_A = [\mathtt{g}(v,u)]_A$ for any terms u, v of sort `Nat`), and constants $0, 1, 2, 3, 4, 5 \in \Sigma_{\mathtt{Nat}}$. Furthermore, R contains two rewrite rules that zero out the first and the second argument of a state fragment of the form $g(u,v)$, respectively:

$$\mathtt{zero1}: \ \mathtt{g}(u,v) \longrightarrow \mathtt{g}(0,v)$$
$$\mathtt{zero2}: \ \mathtt{g}(u,v) \longrightarrow \mathtt{g}(u,0)$$

where u and v are variables of sort `Nat`. On top of this rewrite theory we define the following probabilistic strategy σ in canonical form. (We only give part of its definition, enough to illustrate the main issues that may arise.) First, the rule strategy `r` underlying σ is given by:

$$\begin{aligned}\mathtt{psdrule\ RuleStrat} := \mathtt{given\ state:}\ & f(g(x,y),g(z,t)) \\ \mathtt{is:}\ & \mathtt{zero1} \mapsto 10\ \mathtt{;}\ \mathtt{zero2} \mapsto 2/(x+t)\ .\end{aligned}$$

Also, let `c` be the probabilistic context strategy for rule `zero1`:

$$\begin{aligned}\mathtt{psdcontext\ CtxStrat} := \mathtt{given\ state:}\ & f(g(x,y),g(z,t)) \\ \mathtt{rule:}\ & \mathtt{zero1} \\ \mathtt{is:}\ & f(\odot,g(z,t)) \mapsto (x+y)\ \mathtt{;}\ f(g(x,y),\odot) \mapsto 1/z\ .\end{aligned}$$

Finally, let `s` be the probabilistic substitution strategy for rule `zero2` and context $f(g(x,y),\odot)$, where the state pattern is the same as above, $f(g(x,y),g(z,t))$:

$$\begin{aligned}\mathtt{psdsubst\ SubstStrat} := \mathtt{given\ state:}\ & f(g(x,y),g(z,t)) \\ \mathtt{rule:}\ & \mathtt{zero2} \\ \mathtt{context:}\ & f(g(x,y),\odot) \\ \mathtt{is:}\ & \{u \mapsto z, v \mapsto t\} \mapsto z\ \mathtt{;}\ \{u \mapsto t, v \mapsto z\} \mapsto t\ .\end{aligned}$$

Now, assume that the system is in some concrete state, e.g., $[u]_A = [f(g(2,3), g(4,5))]_A$. The weight associated with the rule label `zero1` is 10 in all states. However, the weight associated with `zero2` is computed by solving the following unification problem:

$$f(g(x,y), g(z,t)) \stackrel{?}{=} f(g(2,3), g(4,5))$$

Due to the commutativity of g, there are four possible solutions to this problem:

$$\theta_1 = \{x \mapsto 2, y \mapsto 3, z \mapsto 4, t \mapsto 5\}, \quad \theta_2 = \{x \mapsto 3, y \mapsto 2, z \mapsto 4, t \mapsto 5\}$$
$$\theta_3 = \{x \mapsto 2, y \mapsto 3, z \mapsto 5, t \mapsto 4\}, \quad \theta_4 = \{x \mapsto 3, y \mapsto 2, z \mapsto 5, t \mapsto 4\}$$

These solutions associate different possible weights to `zero2` in the state $[f(g(2,3), g(4,5))]_A$, making `r` an ill-defined probabilistic rule strategy expression. Namely, θ_1 and θ_2 associate a weight of $\theta_1(2/(x+t)) = \theta_4(2/(x+t)) = 2/7$, θ_2 a weight of $1/4$, and θ_3 a weight of $1/3$.

On the other hand, there are two possible contexts in which the rule `zero1` can be applied, i.e., $[f(\odot, g(4,5))]_A$ and $[f(g(2,3), \odot)]_A$. The weight associated with $[f(\odot, g(4,5))]_A$ is computed by solving the following unification problem:

$$\begin{cases} f(g(x,y), g(z,t)) \stackrel{?}{=} f(g(2,3), g(4,5)) \\ f(\odot, g(z,t)) \stackrel{?}{=} f(\odot, g(4,5)) \end{cases}$$

with the same solutions θ_1, θ_2, θ_3, θ_4 as previously computed for the weight of `zero2`. It follows that the weight associated with $[f(\odot, g(4,5))]_A$ is the same regardless of the substitution used to instantiate it: $\theta_1(x+y) = \theta_2(x+y) = \theta_3(x+y) = \theta_4(x+y) = 5$. To compute the weight of $[f(g(2,3), \odot)]_A$ we must solve the unification problem:

$$\begin{cases} f(g(x,y), g(z,t)) \stackrel{?}{=} f(g(2,3), g(4,5)) \\ f(g(x,y), \odot) \stackrel{?}{=} f(g(2,3), \odot) \end{cases} \tag{6}$$

which has the same solutions θ_1, θ_2, θ_3, θ_4. However, there are two distinct weights associated with $[f(g(2,3), \odot)]_A$, i.e., $\theta_1(1/z) = \theta_2(1/z) = 1/4$ and $\theta_3(1/z) = \theta_4(1/z) = 1/5$, making the probabilistic context strategy expression `c` ill-defined.

Finally, due to the commutativity of g, there are two valid substitutions matching the state fragment of $[f(g(2,3), g(4,5))]_A$ corresponding to the context $[f(g(2,3), \odot)]_A$, i.e., matching the state fragment $g(4,5)$ with the lefthand side $g(u,v)$ of rule `zero2`. These substitutions are $\{u \mapsto 4, v \mapsto 5\}$ and $\{u \mapsto 5, v \mapsto 4\}$. To compute the weight that the probabilistic substitution strategy expression `s` associates to $\{u \mapsto 4, v \mapsto 5\}$, we must first solve the problem (6) which has the same solutions $\theta_1, \theta_2, \theta_3, \theta_4$ as before, and only keep those solutions θ^* that either satisfy $\{u \mapsto z, v \mapsto t\} \circ \theta^* = \{u \mapsto 4, v \mapsto 5\}$ or satisfy $\{u \mapsto t, v \mapsto z\} \circ \theta^* = \{u \mapsto 4, v \mapsto 5\}$. The solutions satisfying the first condition are θ_1 and θ_2, which assign a weight of 4 to $\{u \mapsto 4, v \mapsto 5\}$, whereas the solutions satisfying the second condition are θ_3 and θ_4, which assign a different weight of 5 to the same substitution. A similar issue arises when computing the weight assigned by `s` to the other valid substitution $\{u \mapsto 5, v \mapsto 4\}$. The probabilistic substitution strategy expression `s` is therefore ill-defined.

Well-behaved strategies. In this paragraph we formalize the notion of "well-behaved" probabilistic strategy expressions, which are also the proper "inputs" to our PSMaude tool.

Definition 16. *A probabilistic rule/context/substitution strategy expression is* well-behaved *if it is both well-defined and non-blocking.*

Remark 9. A *uniform* probabilistic rule/context/substitution strategy expression is always well-behaved.

Definition 17. *A "composite" probabilistic rule/context/substitution strategy expression is* well-behaved *if and only if all its constituents are well-behaved.*

Definition 18. *We say that a probabilistic strategy expression σ is* well-behaved *if its underlying (composite) rule, context and substitution strategies are well-behaved.*

5 Probabilistic Strategies and Their Analysis in Maude

We have extended the Maude system with our probabilistic strategy language, and have also implemented in Maude, and integrated into our tool PSMaude, a probabilistic rewrite command and a statistical PCTL model checker that performs the analysis for a given (possibly probabilistic) rewrite theory and probabilistic strategy expressions. The Maude implementation of our probabilistic strategy language and a few examples can be downloaded from `http://heim.ifi.uio.no/~lucianb/psmaude/`.

To specify a probabilistic rewrite rule, the following syntax is used:

`prl [`l`]:` $t(\vec{x})$ `=>` $t(\vec{x},\ \vec{y})$ `with probability` $\vec{y}$ `:= (`$f_1(\vec{x})$ `->` $p_1(\vec{x})$ `;` $\ldots$ `;` $f_n(\vec{x})$ `->` $p_n(\vec{x})$`) .`

which intuitively assigns $f_i(\vec{x})$ to $\vec{y}$ with probability $p_i(\vec{x})$. The keyword `cprl` is used for *conditional* probabilistic rewrite rules. A (possibly object-oriented) module containing such rule declarations must begin with the keyword `(pmod` and end with `endpm)`.

Given a module *`SYSTEM-SPEC`* that specifies a system as a (possibly probabilistic) module, different probabilistic strategy modules can be written that define strategies for *`SYSTEM-SPEC`*. Such modules have the following form:

```
(psmod PSTRAT is protecting SYSTEM-SPEC .          --- import system specification
  state StateSort .                                --- specify sort for system states
  psdrule    RuleStratID    := RuleStratExpr .     --- define rule strategy
  psdcontext ContextStratID := ContextStratExpr . --- define context strategy
  psdsubst   SubstStratID   := SubstStratExpr .    --- define substitution strategy
  psd StratID := < RuleStratID | ContextStratID | SubstStratID > . --- entire strategy
endpsm)
```

Remark 10. The weight expressions in a probabilistic strategy specification represent *relative* weights since the set of possible R/A-matches in a state, whose nondeterministic choice we want to quantify, is only available during execution. We therefore build the concrete probability distributions *on-the-fly* in each state during execution, which we sample to obtain a successor state.

The main idea is that probabilistic strategy declarations, like *`StratID`* above, provide abbreviations for expressions that can be used in a *probabilistic strategy rewrite* command `(prew [`n`]` t `using` σ `.)`, which executes n one-step (probabilistic) rewrites starting from the term t using the probabilistic strategy σ. The unbounded probabilistic rewrite command `(uprew` t `using` σ `.)` rewrites until a deadlock occurs. Furthermore, the command `(prew-once` t `using` σ `.)` can be used for a one-step probabilistic rewrite, being the same as `(prew [1]` t `using` σ `.)`. Finally, `(continue .)` can be used to obtain the result of a one-step probabilistic rewrite under the current strategy, from the last system state obtained with a `prew` or a `prew-once` command.

A probabilistic strategy definition—introduced with the keyword `psd`—associates the probabilistic strategies for rules, contexts, and substitutions, with a probabilistic strategy identifier (on the lefthand side). The probabilistic strategies for rules, contexts, and substitutions are introduced with the keywords `psdrule`, `psdcontext`, and `psdsubst`, respectively. These probabilistic strategy definitions can be conditional, with keywords `cpsdrule`, `cpsdcontext`, and `cpsdsubst`, respectively. There may be several definitions for the same strategy identifier, but they should refer to disjoint cases of the arguments. `[owise]`-annotated strategy expressions can be used to specify how the nondeterminism is resolved when no other strategy definition is applicable. It can thus be easily ensured that a probabilistic strategy resolves *all* nondeterminism in a given system specification, in all possible system states.[7]

Further analysis of a system specification with given probabilistic expressions is possible using our statistical model checking command `(smc` t `|=` φ `using` σ `.)`, where φ is a PCTL formula, and where satisfaction of atomic propositions is defined in a separate *state predicate module*, with the following syntax:

```
(spmod SYSTEM-PRED is protecting SYSTEM-SPEC . --- import system specification
  smcstate StateSort .                         --- specify sort for system states
  psp φ1 ... φn : Sort1 ... SortK .            --- declare parametric state predicates
  var S : StateSort .                          --- and define their semantics
  csat S |= φ1(s1, ..., sK) if f(S, (s1, ..., sK)) .
  ...
endspm)
```

The bounds on the probabilities of returning an erroneous result should be set using the commands `(set type1 error` b_1 `.)`, which sets the upper bound on type I errors (the algorithm returns "false" when the system satisfies the property, also known as "false positives"), and `(set type2 error` b_2 `.)`, that sets the upper bound on type II errors ("false negatives"). By lowering the values of these upper bounds, a higher confidence on the statistical model checking result is achieved, at the cost of generating a larger number of execution samples.

By default, PSMaude uses the same internal parameters for the statistical model checking algorithm as those used by the VeStA tool, namely: *i)* a *stopping probability* of $p_s = 0.1$ for checking unbounded until formulas; *ii)* an *indifference region* defined by $\delta_1 = 0.01$; *iii)* a *tolerance* given by $\delta_2 = 0.01$; and *iv)* a *discount probability* of $p_d = 0.1$ used in the discounting optimization of the algorithm.[8] If needed, these internal parameters can be changed using the following commands: `(set pstop` p_s `.)`, `(set delta1` δ_1 `.)`, `(set delta2` δ_2 `.)` and `(set pdisc` p_d `.)`, respectively.

It is worth pointing out that PSMaude assumes, but does *not* check, that the given probabilistic strategy expressions are well-behaved, i.e., that they are in canonical form, well-defined and non-blocking, and that they quantify all nondeterminism in the "base" model.

Remark 11. PSMaude may detect at run-time *blocking* rule, context and substitution strategy expressions that forcefully lead the system into a deadlock state, in which case it notifies the user with the respective messages:

```
Rule strategy error: One or more rules are enabled, but all have zero probability!

Context strategy error for rule [...]:
One or more contexts are enabled for this rule, but all have zero probability!
```

[7] In its current implementation, PSMaude only supports `[owise]` annotations for rule strategy expressions, as illustrated in the last example of this section.

[8] We refer to [32] for more details about these internal parameters and the algorithm.

```
Substitution strategy error for rule [...] and context: ...
One or more substitutions are enabled for this rule and context, but all have zero probability!
```

Furthermore, PSMaude assumes that all nondeterminism is resolved by the given probabilistic strategy expression. If some nondeterministic choices are left unquantified, they are automatically assigned a zero probability, which may lead to undesired deadlocks.

If the given probabilistic strategy specification fails to meet either one of these requirements, PSMaude may give unexpected results. It is up to the user to ensure that all strategy expressions in her specification meet these requirements.

5.1 Examples

This section illustrates our strategy language by a few simple examples.

Example 1. A *uniform* probabilistic strategy for an arbitrary probabilistic rewrite theory $\mathcal{R}_\pi$ specified as a probabilistic module *SYSTEM-SPEC* can easily be specified using a probabilistic strategy module of the following form[9]:

```
(psmod UNIFORM is protecting SYSTEM-SPEC .
  state StateSort .
  var S : StateSort .  rule R .  context C .
  psdrule       UnifRule := given state: S  is: uniform .
  psdcontext UnifContext := given state: S  rule: R  is: uniform .
  psdsubst     UnifSubst := given state: S  rule: R  context: C  is: uniform  .
  psd UnifStrat := < UnifRule | UnifContext | UnifSubst > .
endpsm)
```

Example 2. We consider a probabilistic variation of the *blackboard game* in [27]. At the beginning of this one-player game, some natural numbers are written on a blackboard. At each step of the game, the player first picks two numbers from the blackboard, *labeling* one of them M and the other N, and then erases them and *tosses a fair coin twice*. If both tosses result in "heads" (which happens with probability $1/4$), she computes the value $K = M^3$; otherwise she computes $K = M^2$ (with the remaining probability of $3/4$). Finally, she computes and writes the value of the expression $(K+N)$ `quo` 2 on the blackboard, where `quo` denotes integer division. As in [27], the aim of the game is to maximize the *expected*[10] final value written on the blackboard.

This game is formalized in the following probabilistic module, that also defines an initial state:

```
(pmod BLACKBOARD is
  protecting RAT .
  sort Blackboard .  subsort Nat < Blackboard .
  op empty : -> Blackboard [ctor] .
  op __ : Blackboard Blackboard -> Blackboard [ctor assoc comm id: empty] .
  vars M N K : Nat .
```

[9] The syntax for declaring the rule and context variables `R` and `C` in this example is not yet implemented in PSMaude. However, such probabilistic strategy modules are still supported by declaring `R` to be of sort `Qid` and `C` to be of sort *StateSort*. (This is because, internally, the sort for the hole variable `[]` is a subsort of *StateSort*.)

[10] Since this game includes several probabilistic choices due to coin tossing—as opposed to the simple blackboard game in [27]—we can only talk about *stochastic outcomes* of player's decisions, and therefore only of *expected* final values.

```
  prl [play]: M N => (K + N) quo 2
                   with probability K := (M * M -> 3/4 ; M * M * M -> 1/4) .

  op initState : -> Blackboard .
  eq initState = 2 3 5 7 11 13 17 .
endpm)
```

Since the multiset union operator `__` is associative and commutative, the choice of the context, i.e., of the *pair* of numbers i and j from the blackboard, is nondeterministic. Furthermore, due to commutativity of `__`, the choice of which substitution to use, $\{\,\texttt{M} \mapsto i,\ \texttt{N} \mapsto j\,\}$ or $\{\,\texttt{M} \mapsto j,\ \texttt{N} \mapsto i\,\}$, is also nondeterministic. Therefore, the above module specifies a probabilistic rewrite theory with both probabilistic and nondeterministic behaviors, where the nondeterministic choices model player's decisions at each step of the game. To quantify all nondeterminism in this model, we fix one possible probabilistic strategy that the player may adopt, formalized in the following probabilistic strategy module:

```
(psmod BLACKBOARD-PROB-STRAT is
  protecting BLACKBOARD .
  state Blackboard .
  var B : Blackboard . vars X Y : Nat .

  psdrule   RuleStrat := given state: B
                                is: (play) -> 1 .

  psdcontext CtxStrat := given state: X Y B
                               rule: play
                                 is: ([] B) -> (1 / (X * Y)) .

  cpsdsubst  SubStrat := given state: X Y B
                               rule: play
                            context: [] B
                                 is: { M <- X, N <- Y } -> 9 ;
                                     { M <- Y, N <- X } -> 1
  if X <= Y .

  psd BlackboardStrat := < RuleStrat | CtxStrat | SubStrat > .
endpsm)
```

The rule strategy `RuleStrat` assigns weight 1 to the only rule `play`. The context strategy `CtxStrat`, selecting in which context the rule `play` applies, assigns for each pair of numbers x and y on the blackboard, the relative weight $1/(x \cdot y)$ to the context that implies that the numbers x and y are replaced by the rule `play`; i.e., it gives a higher weight to contexts corresponding to picking a pair of small numbers to replace. The substitution strategy `SubStrat` selects the rule match $\{\texttt{M} \mapsto x,\ \texttt{N} \mapsto y\}$ with 9 times as high probability as $\{\texttt{M} \mapsto y,\ \texttt{N} \mapsto x\}$ when $x \leq y$, so that the minimum of the two values is more likely to be squared or raised to the third power.

We now explain the strategy `BlackboardStrat` in more detail. For the initial state $[u]_{ACU} = [\texttt{2 3 5 7 11 13 17}]_{ACU}$, `CtxStrat` assigns weights to each valid context as follows. It first matches the state $[u]_{ACU}$ with the state pattern `X Y B` (where `B` can be `empty`), which gives several matches $\theta_1, \ldots, \theta_N$. Then all valid contexts are generated, which in this example have the form $[\odot\ t]_{ACU}$, with $\odot$ identifying the fragment of $[u]_{ACU}$ that matches the lefthand side of rule `play`, and t is the rest of the state. Next, the weight of each valid context is computed. Each context $[\odot\ t]_{ACU}$ is unified with the context pattern `[] B`, giving a unique match $\{\texttt{B} \mapsto t\}$. Of all the matches $\theta_1, \ldots, \theta_N$ obtained above, only those with $\texttt{B} \mapsto t$ are kept.

The weight associated to $[\odot\ t]_{ACU}$ is then computed by instantiating the weight pattern `1 / (X * Y)` with either one of these last substitutions. (For well-definedness, $\theta_i(\texttt{1 / (X * Y)})$ and $\theta_j(\texttt{1 / (X * Y)})$ should be the same, for all θ_i and θ_j with $\theta_i(\texttt{B}) = \theta_j(\texttt{B})$.) For example, for the context $[\mathbb{C}]_{ACU} = [\odot\ \texttt{3 7 11 13 17}]_{ACU}$ two such matches θ_k with $\theta_k(\texttt{B}) = \texttt{3 7 11 13 17}$ exist: $\theta_1 = \{\texttt{X} \mapsto \texttt{2},\ \texttt{Y} \mapsto \texttt{5},\ \texttt{B} \mapsto \texttt{3 7 11 13 17}\}$ and $\theta_2 = \{\texttt{X} \mapsto \texttt{5},\ \texttt{Y} \mapsto \texttt{2},\ \texttt{B} \mapsto \texttt{3 7 11 13 17}\}$. The weight of the context $[\mathbb{C}]_{ACU}$ is then computed as $\theta_1(\texttt{1 / (X * Y)}) = \texttt{1 / 10}$. Similarly, the weight of the context $[\odot\ \texttt{2 5 7 11 17}]_{ACU}$ is $1/(3 \cdot 13) = 1/39$. After computing the weights of all valid contexts, they are normalized to add up to 1, giving a probability distribution from which a context is picked.

The substitution strategy `SubStrat` solves the same matching and unification problems as above, but further refines the set of matches to those that satisfy the condition `X <= Y`. For the context $[\mathbb{C}]_{ACU}$ above the only such match is θ_1. The weight distribution pattern associated with `SubStrat` is then instantiated by θ_1 to obtain a concrete weight distribution over the matches of the lefthand side of rule `play`: $\{\texttt{M} \mapsto \texttt{2},\ \texttt{N} \mapsto \texttt{5}\} \mapsto 9$; $\{\texttt{M} \mapsto \texttt{5},\ \texttt{N} \mapsto \texttt{2}\} \mapsto 1$. By normalizing the associated weights, a probability distribution is obtained, from which a match is picked, e.g., $\eta = \{\texttt{M} \mapsto \texttt{2},\ \texttt{N} \mapsto \texttt{5}\}$ with probability $9/10$.

Finally, a match for the probabilistic variable `K` of rule `play` is sampled from the distribution $\pi_r([\eta]_{ACU}) = \begin{pmatrix} \{\texttt{K} \mapsto \texttt{4}\} & \{\texttt{K} \mapsto \texttt{8}\} \\ 3/4 & 1/4 \end{pmatrix}$.

We now analyze this system specification, whose nondeterminism is quantified by the probability distributions defined by the strategy `BlackboardStrat`. We use unbounded probabilistic rewriting to show possible final states (the outputs are shown as comments):

```
(uprew initState using BlackboardStrat .) --- 13374
(uprew initState using BlackboardStrat .) --- 231549
(uprew initState using BlackboardStrat .) --- 1580
(uprew initState using BlackboardStrat .) --- 8487630
```

We can also simulate a whole game, step by step, under the given strategy:

```
(prew-once initState using BlackboardStrat .) --- 2 7 7 11 13 17
(continue .) --- 5 7 11 13 17
(continue .) --- 5 7 13 69
(continue .) --- 7 69 69
(continue .) --- 59 69
(continue .) --- 1775
```

For statistical model checking, we first define a state predicate `sumGreaterThan(`i`)`, which is true in a state `S` if the sum of all numbers in `S` is strictly larger than i:

```
(spmod BLACKBOARD-PRED is protecting BLACKBOARD .
  smcstate Blackboard . --- sort for system states
  psp sumGreaterThan : Nat . --- declare a parametric state predicate
  var B : Blackboard . var N : Nat .
  csat B |= sumGreaterThan(N) if sum(B) > N . --- define its semantics
  op sum : Blackboard -> Nat . --- auxiliary function
  eq sum(empty) = 0 . eq sum(N B) = N + sum(B) .
endspm)
```

Before running the statistical model checking, we set an upper bound of 0.01 on the probabilities of both type I and type II errors:

```
(set type1 error 0.01 .)          (set type2 error 0.01 .)
```

We then check that the system satisfies the safety property that the sum of the numbers on the blackboard never exceeds 1000000 with probability at least 0.9. This check can be

interpreted from the point of view of an *opponent*, assuming the player is allowed to leave the game at any step and get rewarded with the sum of values on the blackboard at that step. Namely, if the system satisfies the given safety property with high probability, then the opponent can be confident that the player never gets rewarded with an amount greater than one million whenever she is using the above strategy. Indeed, by running:

```
(smc initState |= P>= 0.9 [G ~ sumGreaterThan(1000000)] using BlackboardStrat .)
```

we obtain the following positive result with high confidence, also showing the estimated probability of $1695/1847 \approx 0.9177$ for this safety property to hold:

```
rewrites: 1732143745 in 63814983ms cpu (71069560ms real) (27143 rewrites/second)
Result Bool: true          Number of samples used: 6813855
Confidence: 99 %           Estimated probability: 1695/1847
```

Benchmarking. For a quick benchmarking of our tool, we implemented a Mathematica script that computes the exact probabilities for the above PCTL property to hold in different initial states.[11] We then used our statistical model checker to estimate these probabilities, using the same bounds on the probabilities of type I and type II errors as above, and the same internal parameters p_s, δ_1, δ_2, p_d as in VeStA. Table 1 gives an overview of the exact and estimated probabilities, truncated to 15 decimals[12], showing that the statistical model checking algorithm in [32], that is implemented in our tool, produces quite precise estimates:

Initial state	Exact probability	Estimated probability	Absolute error
`2 3 5 7`	0.999231623745557	0.998736690128136	0.000494933617420
`2 3 5 7 11`	0.994885738360994	0.993863923479516	0.001021814881478
`2 3 5 7 11 13`	0.979742979116798	0.979426096372496	0.000316882744301
`2 3 5 7 11 13 17`	0.913520543990341	0.917704385489984	0.004183841499642

Table 1. Exact vs. estimated probabilities obtained with our tool for the `BLACKBOARD` example.

Remark 12. In the paper [6] we used a prototype version of our tool that, unfortunately, in its approximately 5000 lines of Maude code (including several numerical algorithms), had a subtle bug that affected the pseudo-random number generation, in that it led to the generation of *non-uniform* random variates instead of the desired uniform ones. For this reason, the statistical model checking result reported there for the `BLACKBOARD` example is incorrect. The exact probability for the PCTL property in [6] to hold is 0.590122207084277 (truncated to 15 decimals) which is below the bound of 0.9. We corrected this error, so that the tool now provides reliable statistical model checking results and precise probability estimates. It would of course be very useful to subject our tool to further correctness and perfomance tests, e.g., the ones proposed in [19].

[11] For the initial state `2 3 5 7 11 13 17` our model has a total of 26,402,772 reachable states, of which 19,384,475 are final states.

[12] Appendix A gives the parallel Mathematica code computing the exact probabilities as rational numbers with very large numerators and denominators. (For the initial state `2 3 5 7 11 13 17`, with the same upper bound of 1000000, both the numerator and the denominator have 89452 digits.)

Example 3. The following example shows how the `owise` strategy modifier can be used to ensure that all nondeterminism is probabilistically quantified in all system states. The system specification models a simple counter that starts from 1, and is either increased by 1, increased by 2, multiplied by 2 or multiplied by 4, by the rules `add1`, `add2`, `mul2` and `mul4`, respectively:

```
(pmod COUNTER is protecting NAT .
  var M : Nat .
  rl [add1]: M => M + 1 .
  rl [add2]: M => M + 2 .
  rl [mul2]: M => M * 2 .
  rl [mul4]: M => M * 4 .

  op initState : -> Nat .  eq initState = 1 .
endpm)
```

For this system we specify the following probabilistic strategy which sets uniform context and substitution strategies, and whose rule strategy is uniform whenever the current counter value is below 100, and sets the highest weight 100 to rule `add1` in all the other possible states (when the counter value is larger than 100), and the small weight of 1 to all other rules:

```
(psmod COUNTER-STRAT is protecting COUNTER .  state Nat .
  cpsdrule RuleStrat := given state: M  is: uniform  if M <= 100 .
  psdrule RuleStrat := given state: M
                                is: (add1) -> 100 ; (add2) -> 1 ; (mul2) -> 1 ; (mul4) -> 1
  [owise] .

  psdcontext CtxStrat := given state: M  rule: R  is: uniform .
  psdsubst SubStrat := given state: M  rule: R  context: C  is: uniform  .

  psd CounterStrat := < RuleStrat | CtxStrat | SubStrat > .
endpsm)
```

One possible simulation result for this PSMaude specification from the given initial state `1`, bounded to 30 probabilistic rewrite steps under the strategy `CounterStrat`, is the following:

```
(prew [30] initState using CounterStrat .)

bounded probabilistic rewrite of term: initState
in COUNTER using probabilistic strategy CounterStrat

Rules applied: mul2 mul2 mul2 mul4 add1 add2 mul4 mul2 mul2 add1 mul4 add1 mul2 mul4 add1
    add1 add1 add1 add1 add1 add2 add1 add1 add1 add1 add1 add1 mul2 add1 add1

Result Nat: 268
```

Notice how, from some point on (when the counter value became greater than 100), the rule that has most frequently been applied was `add1`, as expected from the given rule strategy.

6 Formalizing and Analyzing Cloud Computing Policies

In this section we use a cloud computing example to illustrate the usefulness of defining different adversaries for the same (nonprobabilistic) rewrite theory. The "base" rewrite theory defines all possible behaviors of the cloud and its environment, and each probabilistic strategy corresponds to a particular probabilistic *load balancing policy* (and assumptions about the environment). We prove the safety of the "base" model, and use probabilistic simulation and statistical model checking to analyze the QoS of different load balancing policies. The full

PSMaude specification of this example is given in Appendix B, whereas Appendix C provides the full PSMaude specification for one of the more complex load balancing policies that we consider. The PSMaude specifications for all the load balancing policies mentioned in this report are available from the PSMaude homepage `http://heim.ifi.uio.no/lucianb/psmaude/`.

6.1 A Cloud Computing Scenario

We model a typical cloud computing system in which a cloud service delivers services to *service users* and *service providers*. A service provider runs web applications on one or more *virtual machines* (VMs) hosted on *physical servers* in the cloud. Service users then use these web applications via the cloud service. A typical example of a service user is a person that uses an email application provided by a service provider using a cloud computing infrastructure.

Physical servers are grouped together into clusters called *data centers*, which are again grouped into separate geographical *regions*. Since running web applications in regions closer to the users may prove beneficial, we have included a *region selection* filter in our cloud computing architecture shown in Fig. 1. Each region also has a *load balancer* that distributes incoming traffic across the data centers in that region. We focus on load balancing in this example and abstract from other mechanisms (e.g., security protocols, disk storage facilities, etc.). Virtual machines may shut down at any time, due to a failure of the physical server on which they run. These situations are handled by *migrating* virtual machines from a failing physical server to a more reliable one. We assume that the failed virtual machines cannot be recovered, and we therefore only migrate running ones.

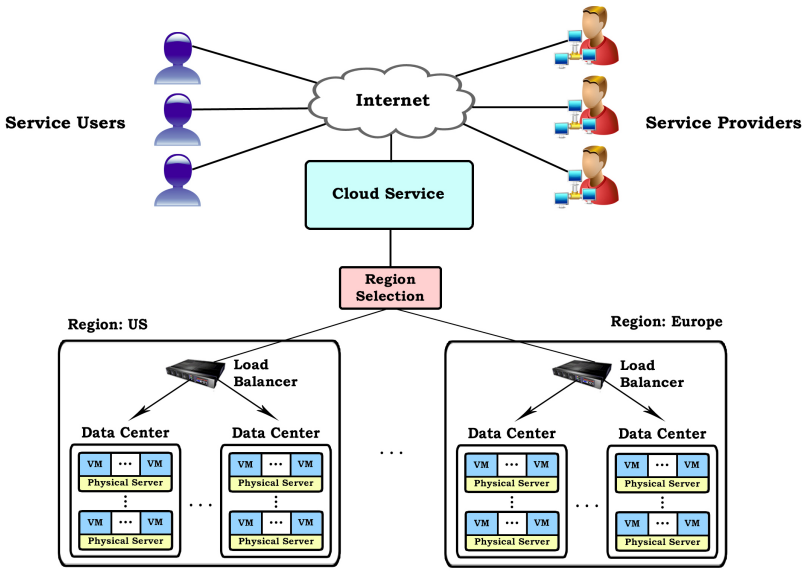

Fig. 1. A cloud computing architecture.

When a service *user* sends an application request to the cloud, the cloud service first identifies the service provider that owns the application, then forwards the request to one of that provider's virtual machines.

Service *providers* may send requests to the cloud service to launch new virtual machines in a particular geographical region. The cloud service forwards these requests to the load balancer associated with that region, which then chooses a physical server to host the virtual machine. Service providers may run up to a given number of virtual machines simultaneously on the cloud, according to their cloud service subscription. Similarly, for each user, there is a bound on how many requests the cloud can process *simultaneously* for that user.

6.2 Formalization of the Cloud Computing System

Static part. We model the cloud computing system as a *hierarchical* object-oriented system. Each main component of the cloud architecture (users, providers, the cloud service, the data centers, and the physical servers) is identified by its IP address of the form `ip(`n`)`, where n is a natural number, except for geographical region objects which are identified by strings, and virtual machines which are identified by names of the form `vm(`p`,`m`)`, with p the identifier of the physical server on which the virtual machine is first launched, and m a unique virtual machine identification number. A sort `Location` denotes the *locations* of the nodes, and the sort `OidMSet` denotes multisets of object identifiers, with `__` as the multiset union operator, and a `size` function giving the number of elements in a multiset:

```
op size : OidMSet -> Nat [memo] .
eq size(noid) = 0 .  eq size(O OIDSET) = 1 + size(OIDSET) .
```

We define the locations of the two regions and of the four users that we consider in our system:

```
ops locUS locEU locA locB locC locD : -> Location [ctor] .
```

We also define the distance in kilometers between locations by a function `distance`, corresponding to the network topology in Figure 2.

```
op distance : Location Location -> Nat [comm memo] .
eq distance(LOC, LOC) = 0 .

*** distance between the two geographical regions
eq distance(locUS, locEU) = 6000 .

*** distances from region A to each user
eq distance(locUS, locA) = 750 . eq distance(locUS, locB) = 375 .
eq distance(locUS, locC) = 750 . eq distance(locUS, locD) = 6047 .

*** distances from region B to each user
eq distance(locEU, locA) = 6047 . eq distance(locEU, locB) = 6375 .
eq distance(locEU, locC) = 6047 . eq distance(locEU, locD) = 750 .

*** distances among the users close to region A
eq distance(locA, locB) = 839 . eq distance(locA, locC) = 1500 .
eq distance(locB, locC) = 839 .

*** distances from the user close to region B, to the users close to A
eq distance(locA, locD) = 6000 . eq distance(locB, locD) = 6419 .
eq distance(locC, locD) = 6185 .
```

We then declare a class for general network nodes, and classes for service users and providers, as subclasses of `Node`:

```
class Node | priority : Nat .
class SUser | location : Location .
class SProvider .
subclass SUser SProvider < Node .
```

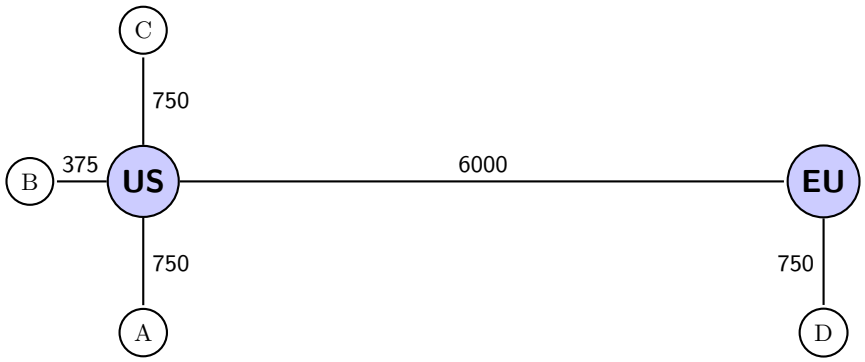

Fig. 2. Network topology used in cloud computing example.

where the attribute `priority` gives the priority for processing requests coming from a node, and `location` denotes the current location of a user. (We abstract away the providers' location, as it has no effect on the functionality of our model.) The class for cloud services is defined by:

```
class CService | status : CServiceDataSet, subscr : CServiceDataSet .
```

where the attributes `status` and `subscr` contain status, and subscription information about both the service users and the service providers, respectively. The sort `CServiceDataSet` is a ','-separated multiset of data of the forms:

- `noReq(ip(`n`), `k`)`: the number of requests that are being processed for user `ip(`n`)` is k;
- `noVM(ip(`m`), `l`)`: the number of virtual machines running for provider `ip(`m`)` is l;
- `maxReq(ip(`n`), `k`)`: the maximum number of requests that the cloud service may simultaneously process for user `ip(`n`)` is k;
- `maxVM(ip(`m`), `l`)`: the maximum number of virtual machines that provider `ip(`m`)` can run simultaneously is l.

A geographical region object has a `location` attribute, a `dataCenters` attribute that contains the set of data center objects in the region, as well as an attribute `vmNo` giving the number of virtual machines (both running and failed) inside the region, i.e., the total workload in the region, and a `vmLoad` attribute giving the total number of user requests that are currently being processed in the region on its virtual machines:

```
class Region | location : Location, dataCenters : Configuration, vmNo : Nat, vmLoad : Nat .
class DCenter | pservers : Configuration .
```

The attribute `pservers` is a multiset of physical server objects, of the class:

```
class PServer | load : Nat, maxLoad : Nat, nextVMID : Nat, vmLoad : Nat, vmFailed : Nat, vms : Configuration .
```

where `load` gives the number of virtual machines currently running on the server; `maxLoad` is the server's capacity; `nextVMID` is a counter for generating fresh IDs for new virtual machines; `vmLoad` is the current total workload over all VMs on the server; `vmFailed` is the number of virtual machines that crashed on the server; and `vms` is a multiset of virtual machine objects, of the class:

```
class VMachine | owner : Oid, vmReq : OidMSet, vmMaxReq : Nat, running : Bool, throughput : Nat .
```

modeling the virtual machines currently running on the physical server, where `owner` represents the ID of the service provider that owns the virtual machine; `vmReq` is the multiset of object IDs of all users whose requests are being resolved on the virtual machine; `vmMaxReq` defines the maximum number of user requests that can be resolved on the virtual machine simultaneously; `running` is a flag saying whether the virtual machine is currently running; and `throughput` is the number of user requests resolved so far on the virtual machine.

We model two types of messages: *i)* user requests `req(ip(`i`), ip(`j`))` from the service user with IP i, to the web application[13] of the service provider with IP j; *ii)* provider requests `launch(ip(`i`),` r`)` from the service provider `ip(`i`)` to the cloud service, to launch a new virtual machine in the geographical region r.

The following shows a possible initial state of our cloud computing model:

```
op initState : -> Configuration .
eq initState = < ip(100) : SUser | location : locA, priority : 1 >
               < ip(101) : SUser | location : locB, priority : 3 >
               < ip(102) : SUser | location : locC, priority : 3 >
               < ip(103) : SUser | location : locD, priority : 5 >
               < ip(200) : SProvider | priority : 2 >
               < ip(201) : SProvider | priority : 5 >
               < ip(300) : CService | status : (noReq(ip(100), 0), noReq(ip(101), 0),
                                                noReq(ip(102), 0), noReq(ip(103), 0),
                                                 noVM(ip(200), 0),  noVM(ip(201), 0)),
                                    subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15),
                                              maxReq(ip(102), 50), maxReq(ip(103), 50),
                                               maxVM(ip(200), 10),  maxVM(ip(201), 10)) >
               <     "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
               < "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 >
               launch(ip(200), "US")  launch(ip(200), "Europe")
               launch(ip(201), "US")  launch(ip(201), "Europe") .
```

with four service users and two service providers, the cloud service, two geographical regions `"US"` and `"Europe"`, four provider requests to launch one virtual machine in each region, and where the constants `dcUS` and `dcEU` define some data centers:

```
ops dcUS dcEU : -> Configuration .
eq dcUS = < ip(400) : DCenter | pservers : psUS1 >  < ip(401) : DCenter | pservers : psUS2 > .
eq dcEU = < ip(500) : DCenter | pservers : psEU1 >  < ip(501) : DCenter | pservers : psEU2 > .
```

each with a set of physical servers with different amounts of resources, that are initially not running any virtual machines:

```
ops psUS1 psUS2 psEU1 psEU2 : -> Configuration .
eq psUS1
 = < ip(4000) : PServer | load : 0, maxLoad : 100, nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none >
   < ip(4001) : PServer | load : 0, maxLoad : 50,  nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none > .
eq psUS2
 = < ip(4002) : PServer | load : 0, maxLoad : 100, nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none >
   < ip(4003) : PServer | load : 0, maxLoad : 50,  nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none > .
eq psEU1
 = < ip(5000) : PServer | load : 0, maxLoad : 200, nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none >
   < ip(5001) : PServer | load : 0, maxLoad : 40,  nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none > .
eq psEU2
 = < ip(5002) : PServer | load : 0, maxLoad : 200, nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none >
   < ip(5003) : PServer | load : 0, maxLoad : 40,  nextVMID : 0, vmLoad : 0, vmFailed : 0, vms : none > .
```

[13] We assume that each service provider runs a single web application.

We also set the value of the maximum number of user requests that can simultaneously be resolved on a virtual machine, which is the same for all machines:

```
op MAXREQ : -> Nat .  eq MAXREQ = 3 .
```

Dynamic part. The following conditional rewrite rule models how the cloud service handles a request from a service user `O` to the web application owned by the service provider `O'`, provided that the service user's new number of requests does not exceed her subscription (the condition `A < B`). This request is forwarded to *any* possible *running* virtual machine owned by `O'` that has enough resources to handle a new user request (the condition `size(OIDSET) < D`). (We therefore implicitly model how a region selection filter nondeterministically picks one of the regions, and then the load balancer associated with that region nondeterministically selects a data center in that region, a physical server inside that data center, as well as a virtual machine to which the user request is forwarded.) After forwarding the request, the user's status is updated in the cloud service, and the selected virtual machine updates its `vmReq` attribute by adding the user's object ID `O` to the multiset `OIDSET`. The workload attribute `vmLoad` for the selected region and for the selected physical server is also updated accordingly. Below, the variable `O''` is of sort `Oid`:

```
vars AS1 AS2 : CServiceDataSet .  vars CSO DCO PSO VMO : Oid .  vars VM PS DC : Configuration .

crl [processUserReq]:
    req(O, O')
    < CSO : CService | status : (noReq(O, A), AS1) , subscr : (maxReq(O, B), AS2) >
    < R : Region |      vmLoad : VMLDR ,
                  dataCenters : < DCO : DCenter |
                     pservers : < PSO : PServer | vmLoad : VMLD ,
                          vms : < VMO : VMachine |
                                  owner : O', running : true, vmReq : OIDSET , vmMaxReq : D >
                                VM > PS > DC >
 => < CSO : CService | status : (noReq(O, A + 1), AS1) >
    < R : Region |      vmLoad : VMLDR + 1 ,
                  dataCenters : < DCO : DCenter |
                     pservers : < PSO : PServer | vmLoad : VMLD + 1 ,
                          vms : < VMO : VMachine | vmReq : OIDSET O >
                                VM > PS > DC >
if A < B /\ size(OIDSET) < D .
```

The following rule models the resolution of a user request on a virtual machine that is running on some physical server in a data center of some geographical region, by removing an object ID from the machine's `vmReq` multiset attribute. The workloads for the selected region and for the selected server are also updated:

```
rl [resolveUserReq]:
   < CSO : CService | status : (noReq(O, A), AS1) >
   < R : Region |      vmLoad : VMLDR ,
                 dataCenters : < DCO : DCenter  |
                    pservers : < PSO : PServer  | vmLoad : VMLD ,
                         vms : < VMO : VMachine |
                                 running : true, vmReq : O OIDSET , throughput : T >
                               VM > PS > DC >
=> < CSO : CService | status : (noReq(O, A - 1), AS1) >
   < R : Region |      vmLoad : VMLDR - 1 ,
                 dataCenters : < DCO : DCenter  |
                    pservers : < PSO : PServer  | vmLoad : VMLD - 1 ,
                         vms : < VMO : VMachine | vmReq : OIDSET , throughput : T + 1 >
                               VM > PS > DC > .
```

The rule below models how the cloud service handles `launch` requests from a service provider, for a new virtual machine in a region `R`, by launching a virtual machine on one of the physical servers, in one of the data centers in `R`—all chosen nondeterministically. The number of virtual machines running in the selected region (`vmNo`) and on the selected server (`load`) are also updated:

```
crl [processProviderReq]:
    launch(O, R)
    < CSO : CService | status : (noVM(O, A), AS1) , subscr : (maxVM(O, B), AS2) >
    < R : Region |        vmNo : VMNO ,
                  dataCenters : < DCO : DCenter |
                     pservers : < PSO : PServer |
                                    load : M , maxLoad : N, nextVMID : NEXTID ,  vms : VM >
                                PS > DC >
 => < CSO : CService | status : (noVM(O, A + 1), AS1) >
    < R : Region |        vmNo : VMNO + 1 ,
                  dataCenters : < DCO : DCenter |
                     pservers : < PSO : PServer | load : M + 1 , nextVMID : NEXTID + 1 ,
                          vms : VM < vm(PSO, NEXTID) : VMachine | owner : O, running : true,
                                     vmReq : noid, vmMaxReq : MAXREQ, throughput : 0 > >
                                PS > DC >
if A < B /\ M < N .
```

Next, we model the possibility of failure of *any* of the virtual machines. The counter `VMFL` of failed machines on the physical server where this event occurs is also updated:

```
rl [failVM]: < PSO : PServer | vmFailed : VMFL,
                               vms : < VMO : VMachine | running : true >  VM >
          => < PSO : PServer | vmFailed : VMFL + 1,
                               vms : < VMO : VMachine | running : false > VM > .
```

The following conditional rule models how the cloud service, *within one of the data centers* of one of the regions (both chosen nondeterministically), migrates a nondeterministically chosen *running* VM `VMO` from a server `PSO` to another one `PSO'`, i.e., shut down VMs are not migrated. This rule is only enabled if the two servers have different failure ratios, if the "source" server is unreliable (`VMFL > 0`), i.e., it has at least one shut down VM, and that migrating the VM to the "destination" server does not exceed that server's capacity (`M' < N'`). The workloads and the number of virtual machines running on the selected servers are also updated correspondingly after the migration:

```
crl [migrateVM]:
    < PSO  : PServer | load : M, vmLoad : VMLD, vmFailed : VMFL,
                       vms : VM < VMO : VMachine | vmReq : OIDSET, running : true > >
    < PSO' : PServer | load : M', vmLoad : VMLD', maxLoad : N', vmFailed : VMFL', vms : VM' >
 => < PSO  : PServer | load : M - 1,  vmLoad : VMLD - VMNOREQ, vms : VM >
    < PSO' : PServer | load : M' + 1, vmLoad : VMLD' + VMNOREQ, vms : VM' < VMO : VMachine | > >
if M' < N' /\ VMFL > 0 /\ failureRatio(VMFL, M) =/= failureRatio(VMFL', M')
/\ VMNOREQ := size(OIDSET) .
```

The function `failureRatio` gives the failure ratio of a server hosting `M` virtual machines, out of which `VMFL` failed. If no machine failed, we make the convention that the failure ratio is 0:

```
op failureRatio : Nat Nat -> Rat .
eq failureRatio(VMFL, M) = if M == 0 then 0 else VMFL / M fi .
```

Finally, both service users and providers can always send new requests to the cloud service. The rule `newUserReq` models how a user sends a new request to the application owned by some

provider, while the rule `newProviderReq` models how a new provider request is sent to launch a new virtual machine in some region:

```
rl [newUserReq]: < O : SUser | >  < O' : SProvider | >
             => < O : SUser | >  < O' : SProvider | >  req(O, O') .
rl [newProviderReq]: < O : SProvider | >  < R : Region | >
                  => < O : SProvider | >  < R : Region | >  launch(O, R) .
```

6.3 Model Checking Safety Properties

To ensure the safety of our cloud system specification, we model check some invariants, using Maude's `search` command. However, before running any analysis, we must ensure that the state space reachable from the initial system state is finite, for the `search` to terminate. We achieve this by extending the `Node` class with a `reqLeft` attribute that denotes the maximum number of requests that a node is allowed to generate:

```
class Node | priority : Nat, reqLeft : Nat .
```

We also update the `newUserReq` and `newProviderReq` rules correspondingly, that become conditional:

```
crl [newUserReq]:
    < O : SUser | reqLeft : N >      < O' : SProvider | >
 => < O : SUser | reqLeft : N - 1 >  < O' : SProvider | >  req(O, O')
if N > 0 .
crl [newProviderReq]:
    < O : SProvider | reqLeft : N >      < R : Region | >
 => < O : SProvider | reqLeft : N - 1 >  < R : Region | >  launch(O, R)
if N > 0 .
```

so that a user/provider can only generate new requests if its associated `reqLeft` attribute is nonzero. This attribute is also decreased with each generated request. In what follows, we model the same infrastructure in each geographical region, with two physical servers per data center, to allow virtual machines to migrate between servers in the same data center. We call the resulting specification `CLOUD-SPEC-SAFETY`, to distinguish it from our base, infinite state space specification defined in Section 6.2, which we simply call `CLOUD-SPEC`.

User Request Safety. We first verify that the cloud service is never processing a larger number of requests from a user `O` than allowed by her subscription (the condition `M > N` in the search below specifies the undesired situation). For this, we use the following initial state, which allows the user `ip(100)` to generate at most 6 requests, and the user `ip(101)` at most 7 requests. The subscription information that the cloud stores about these users is that no more than 5 requests can be resolved simultaneously for user `ip(100)`, and no more than 6 requests for user `ip(101)`. We also allow the provider `ip(200)` to generate one request to launch a virtual machine, on which the user requests will be resolved. We removed provider `ip(201)` from the original initial state, since it is not needed in this scenario:

```
eq initState
 = < ip(100) : SUser | location : locA, priority : 1, reqLeft : 6 >
   < ip(101) : SUser | location : locB, priority : 3, reqLeft : 7 >
   < ip(200) : SProvider | priority : 2, reqLeft : 1 >
   < ip(300) : CService | status : ( noReq(ip(100), 0),  noReq(ip(101), 0),  noVM(ip(200), 0)),
                          subscr : (maxReq(ip(100), 5), maxReq(ip(101), 6), maxVM(ip(200), 2)) >
   <     "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
   < "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 > .
```

By running the following search command to check the previously mentioned safety property:

```
(search [1] in CLOUD-SPEC-SAFETY : initState =>*
  CONFIG < CSO : CService | status : (noReq(O, M), AS1), subscr : (maxReq(O, N), AS2) >
such that M > N .)
```

we obtained the result

```
rewrites: 1615197 in 21841ms cpu (21845ms real) (73950 rewrites/second)
No solution.
```

which confirms the fact that the cloud service never processes more requests from either one of the service users than allowed by their subscription, from the given initial state.

Provider Request Safety. We also verify that the cloud service never creates more virtual machines for a provider `O` than allowed by her subscription (`M > N` in the search below). For this, we use the following initial state, which allows both providers `ip(200)` and `ip(201)` to generate at most 3 requests. However, the subscription information for these providers is that neither one of them can run more than 2 virtual machines simultaneously on the cloud. We did not include any user in this initial state, since we focus on the safety of the cloud service only with respect to the creation of virtual machines.

```
eq initState = < ip(200) : SProvider | priority : 2, reqLeft : 3 >
               < ip(201) : SProvider | priority : 5, reqLeft : 3 >
               < ip(300) : CService | status : ( noVM(ip(200), 0),  noVM(ip(201), 0)),
                                    subscr : (maxVM(ip(200), 2), maxVM(ip(201), 2)) >
               <     "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
               < "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 > .
```

By running the command

```
(search [1] in CLOUD-SPEC-SAFETY : initState =>*
  CONFIG < CSO : CService | status : (noVM(O, M), AS1), subscr : (maxVM(O, N), AS2) >
such that M > N .)
```

we obtain the following result:

```
rewrites: 31403005 in 735590ms cpu (736830ms real) (42690 rewrites/second)
No solution.
```

which proves that the cloud service never runs more virtual machines simultaneously for either one of the service providers than allowed by their subscriptions, for this particular initial state.

6.4 Load Balancing Policies as Probabilistic Strategies

The "base", infinite state space model presented in Section 6.2 describes *all possible* treatments of service provider and service user requests. To achieve possibly better overall performance of the system, at least for the better-paying subscribers, one could think about different *load balancing policies*, so that, e.g., better-paying *users* may get a virtual machine closer to their location with higher probability, or get virtual machines on physical servers with the least workload with high probability, etc. Likewise, a service provider might want to request virtual machines uniformly across the regions to be present all over the world, or might want to request virtual machines in regions with least workload, etc. Within a region, requests could be assigned to servers with small workload to better-paying providers with high probability, or they could be assigned stable (i.e., non-failing) servers with high probability. The point is that these different load balancing policies can be defined by different probabilistic strategies on top of our basic model, whose safety is already verified. In this paper we define three probabilistic strategies which model three different load balancing policies:

i) A strategy that does not take the user locations or the geographical regions into account, i.e., the region to which a pending user request is forwarded is chosen uniformly at random.

ii) A strategy which forwards high priority user requests to the geographical region that is closest to the user, with high probability. This strategy also first processes with high probability requests coming from providers with high priorities.

iii) A strategy that forwards high priority user requests to a geographical region with small virtual machine load, with high probability. High priority provider requests are also processed first, with high probability.

For each of the above strategies, we define two "subcases": *a)* distribute user/provider requests uniformly within the selected/given region, and *b)* with high probability, distribute the requests to virtual machines/physical servers with small workload.

Before defining the strategies corresponding to different load balancing policies, we quantify the nondeterministic choices related to general/environment assumptions about the cloud system. We first define the following probabilistic rule strategy:

```
psdrule RuleStrat := given state: CF
                     is: (resolveUserReq) -> 10 ;
                         (processUserReq) -> 1000  ; (processProviderReq) -> 100 ;
                         (failVM)         -> 1     ; (migrateVM)          -> 1   ;
                         (newUserReq)     -> 100   ; (newProviderReq)     -> 10  .
```

which associates a large (relative) weight of 1000 to processing user requests in this scenario; weight 100 to processing provider requests, and to generating new user requests; weight 10 to resolving user requests, and to generating provider requests; and weight 1 to shutting down virtual machines (modeling a server failure) and to migrating virtual machines from a failing physical server (such that migration does not happen very often, since it is costly). Since `CF` is a variable of sort `Configuration`, this rule strategy can be applied in any possible system state. The above rule strategy therefore quantifies the nondeterministic choice of which rule to apply at each execution step. In what follows, we define the common context and substitution strategies for our load balancing policies, i.e., the context and substitution strategies for all rules, except the ones for `processUserReq` and `processProviderReq`, which differ for each policy.

Remark 13. Due to the hierarchical nature of our cloud model, there are several nondeterministic choices to make when matching the current system state $[u]_A$ with the state pattern in the "`given state`" part of the probabilistic strategy definitions given in this section. Therefore, the weights set to the various context and substitution patterns are actually products of the weights associated to each nondeterministic choice in the state pattern match, at each hierarchical level, provided that these choices are made independently.

Remark 14. Substitution strategies are generally used to specify the relative probabilities of the different permutations of objects and/or messages that match the lefthand side of a rule, due to the `Configuration` union operator being commutative.

To the rule `resolveUserReq` we associate uniform context and substitution strategies:

```
psdcontext CtxStrat := given state: CF  rule: resolveUserReq  is: uniform .
psdsubst SubstStrat := given state: CF  rule: resolveUserReq  context: CTX  is: uniform .
```

where `CTX` is a context variable matching any context. This models that we set no preference as to which user requests the cloud service resolves first; i.e., the object ID `O` matched in this

rule is chosen uniformly at random. Furthermore, the region, the data center, the physical server and the virtual machine on which the user request is resolved are all picked uniformly at random.

We also associate uniform context and substitution strategies with the rule `failVM`:

```
psdcontext CtxStrat := given state: CF  rule: failVM  is: uniform .
psdsubst SubstStrat := given state: CF  rule: failVM  context: CTX  is: uniform .
```

modeling how either one of the virtual machines on either one of the physical servers may crash, uniformly at random.

We set uniform context and substitution strategies to the rule `newProviderReq`:

```
psdcontext CtxStrat := given state: CF  rule: newProviderReq  is: uniform .
psdsubst SubstStrat := given state: CF  rule: newProviderReq  context: CTX  is: uniform .
```

meaning that the service provider `O`, as well as the geographical region `R` in the `launch(O, R)` message generated by the rule, are both selected uniformly at random. However, to the rule `newUserReq` we associate the following non-uniform context strategy, which models the fact that service users are 5 times more likely to send requests to the application of the service provider with IP 201, than to the application of the service provider with IP 200:

```
psdcontext CtxStrat :=
  given state: CF < ip(N) : SUser | ATTS >
               < ip(200) : SProvider | ATTS' >  < ip(201) : SProvider | ATTS'' >
         rule: newUserReq
           is: (CF < ip(201) : SProvider | ATTS'' > []) -> 1 ; --- generate req(ip(N), ip(200))
               (CF < ip(200) : SProvider | ATTS'  > []) -> 5 . --- generate req(ip(N), ip(201))
```

where `ATTS`, `ATTS'` and `ATTS''` are all variables of sort `AttributeSet`, matching any set of object attributes. Therefore, this context strategy models a particular *environment* for the cloud computing system. We also set a uniform substitution strategy for the rule `newUserReq`, since there will always be a single matching substitution:

```
psdsubst SubstStrat := given state: CF  rule: newUserReq  context: CTX  is: uniform .
```

To the rule `migrateVM` we set a context strategy that discourages virtual machine migration between hosts with similar reliabilities, since that would not bring much benefit. (This resolves the nondeterministic choice of which two servers should participate in a migration.)

```
psdcontext CtxStrat :=
  given state: CF < R : Region | ATTS,
                    dataCenters : < DCO : DCenter |
                                    pservers : <  PSO : PServer | load : M,  vmFailed : VMFL,  ATTS'  >
                                               < PSO' : PServer | load : M', vmFailed : VMFL', ATTS'' >
                                               PS
                                  > DC >
         rule: migrateVM
           is: (CF < R : Region | ATTS,
                     dataCenters : < DCO : DCenter | pservers : []  PS > DC >)
               -> (abs(failureRatio(VMFL, M) - failureRatio(VMFL', M'))) .
```

This context strategy models the fact that the probability for two physical servers to take part in a virtual machine migration is large if the absolute difference between their failure ratios is large, i.e., the higher the absolute difference, the more likely that a virtual machine will migrate between them. (The fact that the absolute difference is always nonzero is ensured by the condition of the rule `migrateVM`.) This probabilistic context strategy definition therefore

encourages migration between hosts with different failure ratios, and discourages migration between hosts with similar reliabilities, since that would not bring much benefit. Finally, it implicitly associates a weight of 1, i.e., an uniform distribution over the geographical regions and the different data center objects inside which the virtual machine migration takes place.

We also define a substitution strategy for the rule `migrateVM`, and for the context that identifies two servers participating in a VM migration. This strategy models that, given a pair of two, possibly unreliable servers, the probability for one to be the "source" server of the migration is larger if the server's failure ratio is large, whereas the probability for one to be the "destination" server is larger if the server's failure ratio is small. This strategy therefore encourages migration from a host with high failure ratio to a possibly more reliable host, with small failure ratio. This choice is probabilistic, since we cannot know in advance whether choosing the host with smaller failure ratio as the "destination" server is beneficial in the long term. We therefore also allow the possibility of migrating VMs from the more reliable host, to the host with higher failure ratio, but with small probability. Moreover, we associate a higher probability to migrating a virtual machine with large throughput `T`, and which belongs to a provider with high priority `P`—where the latter is a more important factor, since it is squared in the associated weight expression. Below we prefixed some of the strategy variables by an `X` (e.g., `XVMFL'`) to distinguish them from the corresponding variables in the lefthand side of the rule `migrateVM` that they instantiate (e.g., `VMFL'`).

```
psdsubst SubstStrat :=
  given state: CF < O : SProvider | priority : P >
               < R : Region | ATTS,
                 dataCenters : < DCO : DCenter |
                                 pservers : < XPSO : PServer | ATTS',
                                              load : XM, vmLoad : XVMLD, vmFailed : XVMFL,
                                              vms : < XVMO : VMachine | ATTS'', vmReq : XOIDSET,
                                                      running : true, owner : O, throughput : T >
                                                    XVM
                                            >
                                            < XPSO' : PServer | ATTS''',
                                              load : XM', maxLoad : XN', vmLoad : XVMLD',
                                              vmFailed : XVMFL', vms : XVM' >
                                            PS >
                               DC >
       rule: migrateVM
    context: CF < O : SProvider | priority : P >
             < R : Region | ATTS, dataCenters : < DCO : DCenter | pservers : [] PS > DC >
         is: { PSO  <- XPSO,  M  <- XM,  VMLD  <- XVMLD,  VMFL  <- XVMFL,
               PSO' <- XPSO', M' <- XM', VMLD' <- XVMLD', VMFL' <- XVMFL', N' <- XN',
               VMO <- XVMO, OIDSET <- XOIDSET, VM <- XVM, VM' <- XVM' }
             -> ( P * P * (1 + T)
                    * ((1 + failureRatio(XVMFL, XM)) / (1 + failureRatio(XVMFL', XM'))) ) .
```

We are now ready to define our three load balancing policies, obtained by defining different context and substitution strategies for the rules `processUserReq` and `processProviderReq`. These strategies therefore resolve the nondeterministic choice of which user/provider request to process next, and on which virtual machine/physical server the request should be resolved.

The First Policy: Uniform Region Selection. One first strategy distributes user requests to either one of the running virtual machines in all available regions, selected uniformly at random. It also distributes provider requests uniformly inside the given region, on either one of the available physical servers. The uniform load balancing is formalized by the `SubstStrat1a` probabilistic substitution strategy below:

```
psdcontext CtxStrat1a := given state: CF  rule: processUserReq  is: uniform .
psdsubst SubstStrat1a := given state: CF  rule: processUserReq  context: CTX  is: uniform .

psdcontext CtxStrat1a := given state: CF  rule: processProviderReq  is: uniform .
psdsubst SubstStrat1a := given state: CF  rule: processProviderReq  context: CTX  is: uniform .
```

The resulting probabilistic strategy quantifies all nondeterminism in the cloud system specification, and we denote it by `Strat1a`.

The second strategy distributes requests to a virtual machine/physical server with small load inside the selected/given region with high probability. To the rule `processUserReq` we set the same uniform context strategy as defined by `CtxStrat1a` (but rename it to `CtxStrat1b`), modeling how the user request and the region matched by the rule are picked uniformly at random. We then define a substitution strategy `SubstStrat1b` to model how the load balancer selects with high probability a virtual machine with small workload `size(XOIDSET)` to forward the user request to, where the probability is inversely proportional to $1 +$ `size(XOIDSET)`. Furthermore, using an implicit factor of 1 in the weight expression below, we also model how the data center and the physical server matched by the `processUserReq` rule are picked uniformly at random:

```
psdsubst SubstStrat1b :=
  given state: CF req(XO, XO')
               < XCSO : CService | status :  (noReq(XO, XA), XAS1),
                                   subscr : (maxReq(XO, XB), XAS2) >
               < XR : Region | ATTS, vmLoad : XVMLDR,
                 dataCenters : < XDCO : DCenter |
                                 pservers : < XPSO : PServer | ATTS', vmLoad : XVMLD,
                                              vms : < XVMO : VMachine | ATTS'',
                                                      owner : XO',
                                                    running : true,
                                                      vmReq : XOIDSET,
                                                    vmMaxReq : XD >
                                              XVM >
                                            XPS >
                               XDC >
         rule: processUserReq
      context: CF []
           is: { O <- XO, O' <- XO', CSO <- XCSO, A <- XA, AS1 <- XAS1, B <- XB, AS2 <- XAS2,
                 R <- XR, VMLDR <- XVMLDR, DCO <- XDCO, PSO <- XPSO, VMLD <- XVMLD,
                 VMO:Oid <- XVMO:Oid, OIDSET:OidMSet <- XOIDSET, D <- XD,
                 VM <- XVM, PS <- XPS, DC <- XDC }
               -> (1 / (1 + size(XOIDSET))) .
```

Similarly, to the rule `processProviderReq` we set the same uniform context strategy as defined by `CtxStrat1a` (but rename it to `CtxStrat1b`), modeling how the next provider request to process and the region matched by the rule are picked uniformly at random. The substitution strategy for the rule `processProviderReq` models how, with high probability, the new virtual machine is launched on a physical server with small load `XM` within the given region `XR`, where the probability weight is inversely proportional to $1 +$ `XM`. Using the same reasoning as above, the data center object is also implicitly selected uniformly at random:

```
psdsubst SubstStrat1b :=
  given state: CF launch(XO, XR)
               < XCSO : CService | status :  (noVM(XO, XA), XAS1),
                                   subscr : (maxVM(XO, XB), XAS2) >
               < XR : Region | ATTS, vmNo : XVMNO,
```

```
                dataCenters : < XDCO : DCenter |
                                pservers : < XPSO : PServer | ATTS',
                                             load : XM,
                                             maxLoad : XN,
                                             nextVMID : XNEXTID,
                                             vms : XVM >
                                           XPS >
                              XDC >
        rule: processProviderReq
     context: CF []
          is: { O <- XO, R <- XR, CSO <- XCSO, A <- XA, AS1 <- XAS1, B <- XB, AS2 <- XAS2,
                VMNO <- XVMNO, DCO <- XDCO, PSO <- XPSO, M <- XM, N <- XN, NEXTID <- XNEXTID,
                VM <- XVM, PS <- XPS, DC <- XDC }
              -> (1 / (1 + XM)) .
```

We denote the resulting probabilistic strategy by `Strat1b`.

The Second Policy: High Priority Requests in Close Regions. In this policy, the context strategy definition for the rule `processUserReq` models the fact that user requests with high priorities are selected with high probability (the quadratic factor $\texttt{P}^2$) together with a region `R`, where the probability is also inversely proportional to the distance `distance(LOC, LOC')` between the user and the region `R`:

```
psdcontext CtxStrat2a :=
  given state: CF < O : SUser | priority : P, location : LOC >
               req(O, O') < CSO : CService | ATTS > < R : Region | location : LOC', ATTS' >
         rule: processUserReq
           is: (CF < O : SUser | priority : P, location : LOC > [])
               -> ( (P * P) / (1 + distance(LOC, LOC')) ) .
```

As in the first policy, the substitution strategy for the rule `processUserReq` sets a uniform distribution for the way the load balancer selects one of the data centers, one of the physical servers inside that data center, and one of the virtual machines running on the selected server:

```
psdsubst SubstStrat2a := given state: CF  rule: processUserReq  context: CTX  is: uniform .
```

The rule `processProviderReq` is assigned the following context strategy, which models the fact that requests coming from providers with high priorities are selected with higher probability (the factor $\texttt{P}^2$ in the weight expression below).

```
psdcontext CtxStrat2a :=
  given state: CF < O : SProvider | priority : P >
               launch(O, R) < CSO : CService | ATTS > < R : Region | ATTS' >
         rule: processProviderReq
           is: (CF < O : SProvider | priority : P > []) -> (P * P) .
```

Furthermore, we model that the load balancer launches the virtual machine on either one of the physical servers, in either one of the data centers in the given region `R`, both selected uniformly at random:

```
psdsubst SubstStrat2a := given state: CF  rule: processProviderReq  context: CTX  is: uniform .
```

The resulting probabilistic strategy is denoted `Strat2a`.

The probabilistic strategy `Strat2b` is simply obtained by assigning the context strategy `CtxStrat2a` together with the substitution strategy `SubstStrat1b` to the rules `processUserReq` and `processProviderReq`, but renaming them to `CtxStrat2b` and `SubstStrat2b`, respectively.

The Third Policy: High Priority Requests in Regions with Small Load. In this last strategy, user requests with high priorities are processed with high probability (the quadratic factor `P`^{2} below), and the probability is also inversely proportional to the total virtual machine workload in the selected region. Therefore, with high probability, the region with the smallest virtual machine workload is selected to forward requests to, from users with high priorities:

```
psdcontext CtxStrat3a :=
  given state: CF < O : SUser | priority : P, ATTS >
               req(O, O') < CSO : CService | ATTS' > < R : Region | vmLoad : VMLD, ATTS'' >
         rule: processUserReq
           is: (CF < O : SUser | priority : P, ATTS > []) -> ((P * P) / (1 + VMLD)) .
```

The context strategy for the rule `processProviderReq` is the same as the one given by the context strategy `CtxStrat2a`, picking with high probability requests from providers with high priorities, and setting a uniform distribution over the remaining nondeterministic choices. We rename this context strategy to `CtxStrat3a` and call the resulting full probabilistic strategy `Strat3a`.

Finally, the probabilistic strategy `Strat3b` is obtained by assigning the context strategy `CtxStrat3a` together with the substitution strategy `SubstStrat1b` to the rules `processUserReq` and `processProviderReq`, but renaming them to `CtxStrat3b` and `SubstStrat3b`, respectively.

6.5 Simulation under Different Execution Policies

In what follows, we run probabilistic simulations of our system with respect to the adversaries defined by the probabilistic strategies `Strat2b` and `Strat3b`. Under both these policies, the cloud always processes with higher probability the requests with higher priority, from either users or providers. Furthermore, in both policies, provider requests to launch a virtual machine in a *given* region are forwarded to a data center selected uniformly at random, and then with high probability to a server with small load, i.e., with few virtual machines running on it. The main difference between these policies is in the way the cloud service selects the region in which to forward user requests, i.e., either by selecting with high probability the region that is closer to the user (in `Strat2b`), or by picking with high probability the region with small virtual machine load (in `Strat3b`). Under both these policies, after a region is selected, a data center and a server are chosen uniformly at random, and then a virtual machine with small workload is selected with high probability.

We compare the *quantitative effects* of these two cloud policies using the following notion of "computational effort" for the cloud service. This can be calculated based on the cost of a single request from a user U, dealt with at a virtual machine running on a physical server S in a region R, which is defined by:

$$cost(U, R, S) = k \cdot \frac{\#noTasks(S)}{capacity(S)} + q \cdot distance(U, R)$$

where $noTasks(S)$ is the number of requests currently being processed on the server S, $capacity(S)$ is the maximum number of requests that the server S can process simultaneously, and k, q are some positive weights. If no virtual machine is running on the server S, then we set the above cost to 0. The value $noTasks(S)$ is calculated by summing up the current workloads of all virtual machines running on the server S, while the capacity of a server is calculated as `MAXREQ` $\cdot\, noVMs(S)$ where $noVMs(S)$ is the number of virtual machines currently running on S. (This is because in our model each virtual machine can process at most `MAXREQ` user requests simultaneously.)

The first term of the above sum is therefore proportional to the relative load of the server at the time when the user request is processed—the larger this relative load, the more time it takes the cloud service to process the new user request. The second term is proportional to the distance between the user and the region R, and can be seen as an estimation of the time it takes to transfer the data from the user to the cloud service, and back. The total sum estimates the cost for the cloud service to resolve the user request on a particular server S in a region R.

The cost of all requests from a user U is obtained by multiplying the above cost with the total *number* of requests from U being processed on each virtual machine of the server S, and then summing up over all physical servers S, and all regions R:

$$costAll(U) = \sum_{R \text{ is a region}} \sum_{S \text{ is server in } R} \left[cost(U,R,S) \cdot \sum_{v \text{ is a VM at } S} noReq(U,v) \right]$$

where $noReq(U,v)$ is the number of requests from user U that are currently being processed at virtual machine v. Finally, the current cost of requests from all users is obtained by summing up over all users in the state:

$$costCloud = \sum_{U \text{ is a user}} costAll(U)$$

which estimates the total "computational effort" for the cloud system to resolve all user requests at a given system state. For both simulations below, we picked the values $k = q = 1$.

For efficiency reasons, instead of recomputing the *costCloud* function above in each system state, we can include the current cumulated computational effort $e \in \mathbb{Q}$ as a term `eff(`e`)` in the global state, i.e.,

```
sort Effort .  subsort Effort < Configuration .  op eff : Rat -> Effort .
```

and update this term after each application of the `processUserReq` rule. We therefore slightly change this rule as follows, where the added parts are emphasized, and where `EFFORT` is a variable of sort `Rat`.

```
crl [processUserReq]:
    eff(EFFORT)  < O : SUser | location : LOC >
    req(O, O')
    < CSO : CService | status : (noReq(O, A), AS1), subscr : (maxReq(O, B), AS2) >
    < R : Region |    location : LOC',
                      vmLoad : VMLDR,
                dataCenters : < DCO : DCenter |
                   pservers : < PSO : PServer | load : M,  vmLoad : VMLD,
                        vms : < VMO : VMachine |
                                  owner : O', running : true, vmReq : OIDSET, vmMaxReq : D >
                              VM > PS > DC >
 => eff(EFFORT + DISTFACTOR * distance(LOC, LOC') + (LOADFACTOR * VMLD) / (1 + M))  < O : SUser | >
    < CSO : CService | status : (noReq(O, A + 1), AS1) >
    < R : Region |      vmLoad : VMLDR + 1,
                dataCenters : < DCO : DCenter |
                   pservers : < PSO : PServer | vmLoad : VMLD + 1,
                        vms : < VMO : VMachine | vmReq : OIDSET O >
                              VM > PS > DC >
if A < B /\ size(OIDSET) < D .
```

Furthermore, we add the model parameters `LOADFACTOR` and `DISTFACTOR`, corresponding to the above mentioned weights k and q, respectively:

```
ops LOADFACTOR DISTFACTOR : -> Rat .  eq LOADFACTOR = 1 .  eq DISTFACTOR = 1 .
```

Finally, we must include the term `eff(0)` in any initial state we consider, since we assume that the cloud service starts from a "fresh" state, in which no user requests were resolved.

Since we changed the `processUserReq` rule, the context and substitution strategies that `Strat2b` and `Strat3b` assign to this rule must also be changed. The context strategy `CtxStrat2b` (which is the same as `CtxStrat2a`) is changed as follows, where the new parts are emphasized:

```
psdcontext CtxStrat2b :=
  given state: CF eff(EFFORT) < O : SUser | priority : P, location : LOC >
               req(O, O') < CSO : CService | ATTS > < R : Region | location : LOC', ATTS' >
         rule: processUserReq
           is: (CF []) -> ( (P * P) / (1 + distance(LOC, LOC')) ) .
```

Similarly, the probabilistic context strategy `CtxStrat3b` (which is the same as `CtxStrat3a`) is changed as follows:

```
psdcontext CtxStrat3b :=
  given state: CF eff(EFFORT) < O : SUser | priority : P, ATTS >
               req(O, O') < CSO : CService | ATTS' > < R : Region | vmLoad : VMLD, ATTS'' >
         rule: processUserReq
           is: (CF []) -> ((P * P) / (1 + VMLD)) .
```

Finally, the (identical) substitution strategies `SubstStrat2b` and `SubstStrat3b` (which are the same as `SubstStrat1b`) are also changed correspondingly:

```
psdsubst SubstStrat2b :=
  given state: CF  eff(XEFFORT) < XO : SUser | location : XLOC, ATTS''' >  req(XO, XO')
               < XCSO : CService | status :  (noReq(XO, XA), XAS1),
                                   subscr : (maxReq(XO, XB), XAS2) >
               < XR : Region | ATTS, vmLoad : XVMLDR, location : XLOC',
                 dataCenters : < XDCO : DCenter |
                                 pservers : < XPSO : PServer | ATTS', load : XM, vmLoad : XVMLD,
                                              vms : < XVMO : VMachine | ATTS'',
                                                      owner : XO',
                                                      running : true,
                                                      vmReq : XOIDSET,
                                                      vmMaxReq : XD >
                                                    XVM >
                                            XPS >
                               XDC >
         rule: processUserReq
      context: CF []
           is: { O <- XO, O' <- XO', CSO <- XCSO, A <- XA, AS1 <- XAS1, B <- XB, AS2 <- XAS2,
                 R <- XR, VMLDR <- XVMLDR, DCO <- XDCO, PSO <- XPSO, VMLD <- XVMLD,
                 VMO:Oid <- XVMO:Oid, OIDSET:OidMSet <- XOIDSET, D <- XD,
                 VM <- XVM, PS <- XPS, DC <- XDC,
                 EFFORT <- XEFFORT, LOC <- XLOC, LOC' <- XLOC', M <- XM }
               -> (1 / (1 + size(XOIDSET))) .
```

In what follows, we use the following initial state for our cloud system, which is the same as the initial state in Section 6.2 after removing all four messages in the state, and adding the `eff(0)` term:

```
eq initState = < ip(100) : SUser | location : locA, priority : 1 >
               < ip(101) : SUser | location : locB, priority : 3 >
               < ip(102) : SUser | location : locC, priority : 3 >
               < ip(103) : SUser | location : locD, priority : 5 >
```

```
< ip(200) : SProvider | priority : 2 >
< ip(201) : SProvider | priority : 5 >
< ip(300) : CService | status : (noReq(ip(100), 0), noReq(ip(101), 0),
                                 noReq(ip(102), 0), noReq(ip(103), 0),
                                  noVM(ip(200), 0),  noVM(ip(201), 0)),
                       subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15),
                                 maxReq(ip(102), 50), maxReq(ip(103), 50),
                                  maxVM(ip(200), 10),  maxVM(ip(201), 10)) >
< "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
< "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 >
eff(0) .
```

We first simulate the execution of our *infinite* state space cloud system under the policy defined by `Strat2b`, using our probabilistic rewrite command bounded by 1000 rewrite steps:

```
(prew [1000] initState using Strat2b .)
```

This returns a result of the following form, also showing the rules that were applied:

```
rewrites: 13655875 in 371971ms cpu (372037ms real) (36712 rewrites/second)
bounded probabilistic rewrite of term: initState
in CLOUD-SPEC using probabilistic strategy Strat2b

Rules applied: newUserReq newUserReq newUserReq newUserReq newUserReq
    newProviderReq processProviderReq processUserReq processUserReq
    processUserReq resolveUserReq processUserReq newUserReq newUserReq ...
    newUserReq newUserReq failVM newUserReq newUserReq newUserReq newUserReq ...
    newUserReq processUserReq migrateVM newUserReq migrateVM newProviderReq ...
    newUserReq newUserReq newUserReq newUserReq newProviderReq newProviderReq

Result Configuration: eff(5154483/20)

< ip(300) : CService | status : (noReq(ip(100), 6), noReq(ip(101), 14), noReq(ip(102), 23), noReq(ip(103), 17),
                                 noVM(ip(200), 10), noVM(ip(201), 10)),
                       subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15), maxReq(ip(102), 50), maxReq(ip(103), 50),
                                 maxVM(ip(200), 10), maxVM(ip(201), 10)) >

launch(ip(200),"Europe") ... launch(ip(200),"Europe")   launch(ip(200),"US") ... launch(ip(200),"US")
launch(ip(201),"Europe") ... launch(ip(201),"Europe")   launch(ip(201),"US") ... launch(ip(201),"US")
req(ip(100),ip(200)) ... req(ip(100),ip(200))   req(ip(100),ip(201)) ... req(ip(100),ip(201))
req(ip(101),ip(200)) ... req(ip(101),ip(200))   req(ip(101),ip(201)) ... req(ip(101),ip(201))
req(ip(102),ip(200)) ... req(ip(102),ip(200))   req(ip(102),ip(201)) ... req(ip(102),ip(201))
req(ip(103),ip(200)) ... req(ip(103),ip(200))   req(ip(103),ip(201)) ... req(ip(103),ip(201))

< "Europe" : Region | location : locEU, vmLoad : 21, vmNo : 7,
   dataCenters : (< ip(500) : DCenter |
                    pservers : (< ip(5000) : PServer | load : 4, maxLoad : 200, nextVMID : 2, vmFailed : 0, vmLoad : 12,
                                  vms : (< vm(ip(5000), 0) : VMachine | owner : ip(200), running : true,
                                            throughput : 12, vmMaxReq : 3, vmReq : (ip(102) ip(103) ip(103)) > ...
                                          < vm(ip(5001), 1) : VMachine | owner : ip(200), running : true,
                                            throughput : 8, vmMaxReq : 3, vmReq : (ip(100) ip(102) ip(102)) >) >
                                < ip(5001) : PServer | load : 1, maxLoad : 40, nextVMID : 3, vmFailed : 1, vmLoad : 3,
                                  vms : < vm(ip(5001), 2) : VMachine | owner : ip(201), running : false,
                                            throughput : 1, vmMaxReq : 3, vmReq : (ip(102) ip(103) ip(103))> >) > ... ) >

< "US" : Region | location : locUS, vmLoad : 39, vmNo : 13,
  dataCenters :(< ip(400): DCenter |
                  pservers : (< ip(4000) : PServer | load : 3, maxLoad : 100, nextVMID : 4, vmFailed : 1, vmLoad : 9,
                                vms : (< vm(ip(4000), 0) : VMachine | owner : ip(200), running : false,
                                          throughput : 1, vmMaxReq : 3, vmReq : (ip(102) ip(103) ip(103)) >
                                        < vm(ip(4000), 2) : VMachine | owner : ip(200), running : true,
                                          throughput : 2, vmMaxReq : 3, vmReq :(ip(100) ip(100) ip(102)) >
                                        < vm(ip(4000), 3) : VMachine | owner : ip(200), running : true,
                                          throughput : 1, vmMaxReq : 3, vmReq :(ip(101) ip(101) ip(101)) >) >
```

```
                              < ip(4001) : PServer | load : 4, maxLoad : 50, nextVMID : 3, vmFailed : 0, vmLoad : 12,
                                vms : (< vm(ip(4000), 1) : VMachine | owner : ip(201), running : true,
                                          throughput : 0, vmMaxReq : 3, vmReq :(ip(101) ip(101) ip(101)) >
                                       ... ) >) > ... ) >

< ip(100) : SUser | location : locA, priority : 1 >  < ip(101) : SUser | location : locB, priority : 3 >
< ip(102) : SUser | location : locC, priority : 3 >  < ip(103) : SUser | location : locD,priority : 5 >
< ip(200) : SProvider | priority : 2 >  < ip(201) : SProvider | priority : 5 >
```

with a cumulated cloud cost of $5154483/20 = 257724.15$. In this result we noticed that the users sent more requests to the application of the provider with IP 201, as specified by the probabilistic strategy. We also noticed that several virtual machine migrations took place in `"Europe"`—e.g., `vm(ip(5001, 1))` has been migrated from server `ip(5001)` to server `ip(5000)`—due to the server `ip(5001)` starting to fail, as shown by its `vmFailed` attribute and by the `vm(ip(5001), 2)` virtual machine that crashed on this server. A virtual machine migration also took place in the `"US"` region—i.e., `vm(ip(4000, 1))` has been migrated from server `ip(4000)` to server `ip(4001)`—due to the server `ip(4000)` starting to fail, as shown by its `vmFailed` attribute and by the `vm(ip(4000), 0)` virtual machine that crashed on this server.

We next simulate the cloud system under the second probabilistic strategy:

```
rewrites: 13817080 in 384485ms cpu (384562ms real) (35936 rewrites/second)
bounded probabilistic rewrite of term: initState
in CLOUD-SPEC using probabilistic strategy Strat3b

Rules applied: newUserReq newUserReq newUserReq newUserReq newUserReq
    newProviderReq processProviderReq processUserReq processUserReq
    processUserReq resolveUserReq processUserReq newUserReq newUserReq ...
    newUserReq processUserReq failVM newUserReq newUserReq newUserReq ...
    processUserReq newUserReq newUserReq newUserReq migrateVM newUserReq failVM ...
    newUserReq newUserReq newUserReq newUserReq newUserReq newProviderReq newProviderReq

Result Configuration: eff(43071484/105)

< ip(300) : CService | status : (noReq(ip(100), 4), noReq(ip(101), 12), noReq(ip(102), 14), noReq(ip(103), 28),
                         noVM(ip(200), 10), noVM(ip(201), 10)),
                       subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15), maxReq(ip(102), 50), maxReq(ip(103), 50),
                         maxVM(ip(200), 10), maxVM(ip(201), 10)) >

launch(ip(200),"Europe") ... launch(ip(200),"Europe")   launch(ip(200),"US") ... launch(ip(200),"US")
launch(ip(201),"Europe") ... launch(ip(201),"Europe")   launch(ip(201),"US") ... launch(ip(201),"US")
req(ip(100),ip(200)) ... req(ip(100),ip(200))   req(ip(100),ip(201)) ... req(ip(100),ip(201))
req(ip(101),ip(200)) ... req(ip(101),ip(200))   req(ip(101),ip(201)) ... req(ip(101),ip(201))
req(ip(102),ip(200)) ... req(ip(102),ip(200))   req(ip(102),ip(201)) ... req(ip(102),ip(201))
req(ip(103),ip(200)) ... req(ip(103),ip(200))   req(ip(103),ip(201)) ... req(ip(103),ip(201))

< "Europe" : Region | location : locEU, vmLoad : 16, vmNo : 6,
  dataCenters : (< ip(500) : DCenter |
                   pservers : (< ip(5000) : PServer | load : 2, maxLoad : 200, nextVMID : 2, vmFailed : 1, vmLoad : 6,
                                 vms : (< vm(ip(5000), 0) : VMachine | owner : ip(200), running : false,
                                           throughput : 3, vmMaxReq : 3, vmReq : (ip(101) ip(102) ip(103)) >
                                       < vm(ip(5001), 0) : VMachine | owner : ip(201), running : true,
                                           throughput : 8, vmMaxReq : 3, vmReq : (ip(102) ip(103) ip(103)) > ) >
                               < ip(5001) : PServer | load : 2, maxLoad : 40, nextVMID : 2, vmFailed : 1, vmLoad : 4,
                                 vms : (< vm(ip(5000), 1) : VMachine | owner : ip(200), running : true,
                                           throughput : 8, vmMaxReq : 3, vmReq : (ip(100) ip(102) ip(103)) >
                                       < vm(ip(5001), 1) : VMachine | owner : ip(200), running : false,
                                           throughput : 0, vmMaxReq : 3, vmReq : ip(102) >) >) > ... ) >

< "US" : Region | location : locUS, vmLoad : 42, vmNo : 14,
  dataCenters : (< ip(401) : DCenter |
                   pservers : (< ip(4002) : PServer | load : 7, maxLoad : 100, nextVMID : 4, vmFailed : 0, vmLoad : 21,
                                 vms : (< vm(ip(4002), 0) : VMachine | owner : ip(201), running : true,
                                           throughput : 1, vmMaxReq : 3, vmReq : (ip(102) ip(102) ip(102)) > ...
```

```
                    < vm(ip(4003), 0) : VMachine | owner : ip(201), running : true,
                      throughput : 3, vmMaxReq : 3, vmReq : (ip(101) ip(102) ip(103)) >
                    < vm(ip(4003), 1) : VMachine | owner : ip(200), running : true,
                      throughput : 2, vmMaxReq : 3, vmReq : (ip(101) ip(102) ip(103)) >
                    < vm(ip(4003), 2) : VMachine | owner : ip(201), running : true,
                      throughput : 1, vmMaxReq : 3,  vmReq :(ip(101) ip(101) ip(103)) >) >
          < ip(4003) : PServer | load : 1, maxLoad : 50, nextVMID : 4, vmFailed : 1, vmLoad : 3,
            vms : < vm(ip(4003), 3) : VMachine | owner : ip(201), running : false,
                    throughput : 0, vmMaxReq : 3, vmReq : (ip(103) ip(103) ip(103)) > >) > ... ) >

< ip(100) : SUser | location : locA, priority : 1 >  < ip(101) : SUser | location : locB, priority : 3 >
< ip(102) : SUser | location : locC,priority : 3 >  < ip(103): SUser | location : locD,priority : 5 >
< ip(200) : SProvider | priority : 2 >  < ip(201) : SProvider | priority : 5 >
```

with a cumulated cost of the cloud of $43071484/105 \approx 410204.61$, which is larger than the one obtained under the previous probabilistic strategy. We again noticed that the users sent more requests to the application of the provider with IP 201, as specified by the probabilistic strategy. Furthermore, in the `"Europe"` region, at least two virtual machine migrations took place between the failing servers `ip(5000)` and `ip(5001)`, and several virtual machines were also migrated in `"US"` region, from the failing server `ip(4003)` to the "healthy" server `ip(4002)`. It therefore seems that the computational effort of the cloud service under the execution policy `Strat3b` attains larger values than under `Strat2b`. This is because the latter always resolves user requests on the closest geographical region, avoiding time delays due to data transfers over the Internet.

Remark 15. Even though such comparison may seem straightforward, we have provided this example to show how our methodology can be used to formalize and compare the quantitative effects of any two *arbitrary* execution policies of a probabilistic distributed system defined by a (infinite state space) PSMaude model.

The two graphs in Figure 3 show how the total cost function *costCloud* evolved over the 1000 probabilistic rewrite steps (the horizontal axis) of the above simulations to reach the final values of 257724.15 and ≈ 410204.61 under the policies defined by `Strat2b` and `Strat3b`, respectively. (We used the same random seed 0 for both simulations.) The way we obtained the graph corresponding to `Strat2b` was by first running the command

```
(prew-once initState using Strat2b .)
```

followed by a series of 999 `(continue .)` commands. (The graph for the `Strat3b` strategy was obtained similarly.) The PSMaude output was then processed using the Python script in Appendix D, generating two `.csv` files with the series of values for the *costCloud* function corresponding to each strategy, which could then easily be plotted. The graph with smaller values in Figure 3 corresponds to `Strat2b` which, as mentioned above, seems to imply that `Strat2b` is a better policy in terms of the total costs.

To get a better idea of how the evolution of the *costCloud* function is different from one policy to another, we also ran a series of 30 simulations under each strategy (with random seeds ranging from 1 to 30). Instead of plotting all obtained trajectories, Figure 4 shows the *mean trajectories* under the two policies, as well as the bounds given by two sample standard deviations above and below these mean trajectories. Since we generated 30 sample trajectories, it follows by direct calculation from the Saw-Yang-Mo[14] inequality [31] that more

[14] We cannot use Chebyshev's inequality since neither the population mean nor the standard deviation of the cloud cost at each simulation step is known. However, we estimated these values by the corresponding sample mean and sample standard deviation, allowing us to use the Saw-Yang-Mo inequality instead.

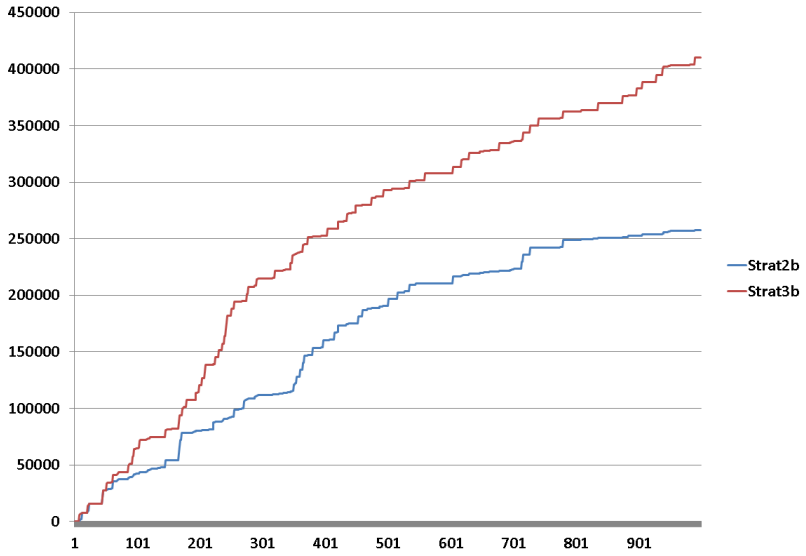

Fig. 3. Comparison of the total cloud cost with each rewrite step, for the two policies.

than approximately 75% of the trajectories of the *costCloud* function lie within the bounds shown in Figure 4. This, together with our simulation results, seem to support the idea that, on average, `Strat2b` is indeed a better policy in terms of the total costs. However, this should be rigorously checked using statistical hypothesis testing, i.e., by means of statistical model checking. Since the state space of the cloud model we considered so far is infinite, we cannot directly use the statistical model checking algorithm implemented in PSMaude, as it requires the state space to be finite [32]. In the next section we therefore set bounds on the total number of requests that users and providers can send to the cloud, as also done in Section 6.3, making the state space of our model finite. This allows us to use PSMaude to statistically model check properties of the evolution of the *costCloud* function under each of the two probabilistic strategies.

6.6 Statistical Model Checking of the Cloud System

As done in Section 6.3, we first ensure that the reachable state space from the initial state is finite, by setting a limit on the number of requests that both users and providers can generate, and modifying the `newUserReq` and `newProviderReq` rules correspondingly. We therefore use the following initial state, where users can send at most 10 requests to the cloud, and providers can send at most 20 requests:

```
eq initState = < ip(100) : SUser | location : locA, priority : 1, reqLeft : 10 >
               < ip(101) : SUser | location : locB, priority : 3, reqLeft : 10 >
               < ip(102) : SUser | location : locC, priority : 3, reqLeft : 10 >
               < ip(103) : SUser | location : locD, priority : 5, reqLeft : 10 >
               < ip(200) : SProvider | priority : 2, reqLeft : 20 >
               < ip(201) : SProvider | priority : 5, reqLeft : 20 >
               < ip(300) : CService | status : (noReq(ip(100), 0), noReq(ip(101), 0),
                                                noReq(ip(102), 0), noReq(ip(103), 0),
                                                noVM(ip(200), 0), noVM(ip(201), 0)),
```

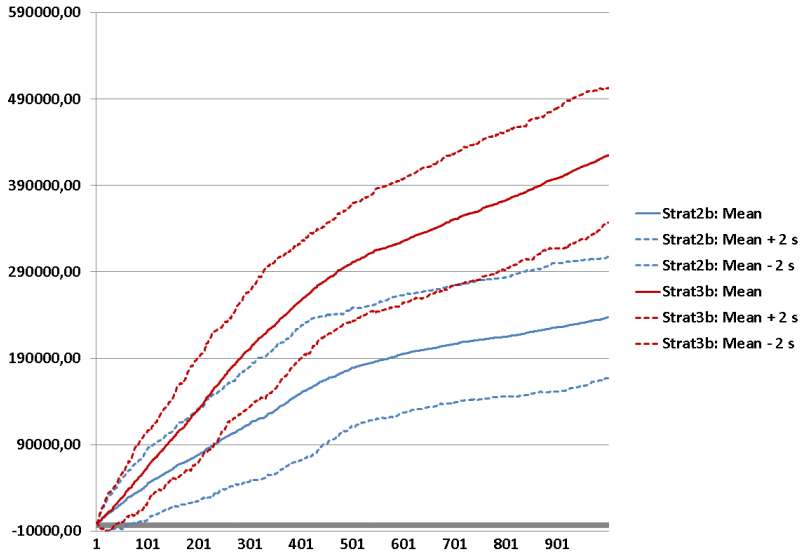

Fig. 4. Mean trajectories of the total cloud cost for the two policies.

```
                    subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15),
                              maxReq(ip(102), 50), maxReq(ip(103), 50),
                              maxVM(ip(200), 10), maxVM(ip(201), 10)) >
< "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
< "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 >
eff(0) .
```

The context strategies for the rules `processUserReq` and `processProviderReq`, as well as the substitution strategy for the rule `migrateVM` are also slightly adjusted for both the `Strat2b` and the `Strat3b` strategies, so that any occurrence of an object of class `SUser` or `SProvider` also includes the newly added `reqLeft` attribute.

Before running the statistical model checking of the resulting finite state space cloud model, we first run some simulations to get an idea of how the *costCloud* function evolves for this model, under each probabilistic strategy. We do this since it is expected that, by setting bounds on the total number of requests that users and providers can send, the evolution of the *costCloud* function should also change (compared with the graphs in Figures 3 and 4).

This time we only simulate 200 probabilistic rewrite steps, since the *costCloud* function seems to remain constant beyond this point. This behavior is likely due to the users and the providers exhausting (with high probability) the maximum number of requests that they can send in the first 200 probabilistic rewrite steps under the two strategies. The simulation result is shown in Figure 5, where we used a random seed of 0 for both graphs. As expected, the final values attained by the *costCloud* function under the `Strat2b` and `Strat3b` strategies are smaller than for the infinite state space model simulated for 1000 probabilistic rewrite steps, i.e., $190279/2 = 95139.5$ and $276437/2 = 138218.5$, respectively. Therefore, it still seems that, in this finite state space model, the *costCloud* function attains smaller values under the `Strat2b` strategy than under `Strat3b`.

We also ran a series of 30 simulations for each policy and plotted the mean trajectories with their "two standard deviations" bounds in Figure 6. Again, even though our model has slightly been changed to have a finite state space, we notice the same behavior as before,

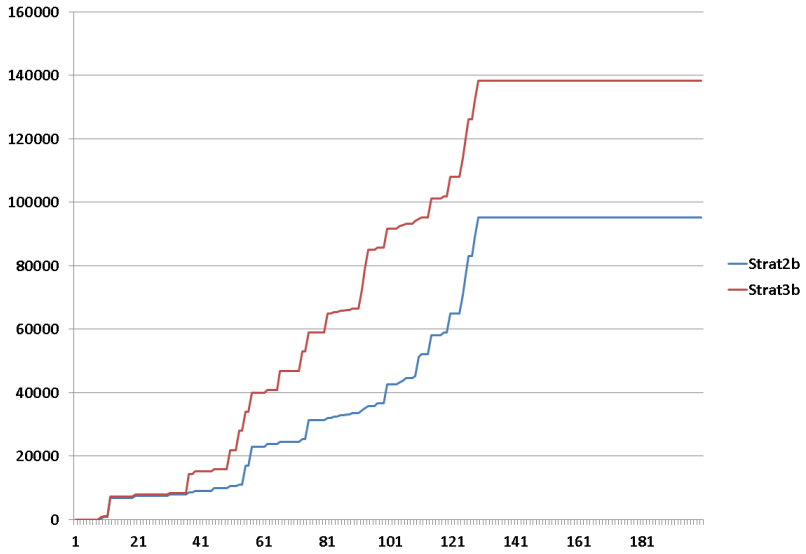

Fig. 5. Comparison of the total cloud cost with each rewrite step for the finite state space cloud model under the two policies.

i.e., on average, `Strat2b` is a better policy in terms of the total costs. More precisely, with the Saw-Yang-Mo inequality in mind, Figure 6 supports the idea that more than 75% of the *costCloud* trajectories under the `Strat2b` policy eventually go beyond the value of about 45000, but never exceed 145000. On the other hand, based on the simulation results we could argue that more than 75% of the *costCloud* trajectories under the `Strat3b` policy eventually go beyond the value of about 92000 and never exceed 166000. Together with the fact that the *costCloud* function can only increase (with each application of the `processUserReq` rule), it seems that, in the long run, the minimum cloud cost under the `Strat3b` policy is somewhere around 100000, whereas under the `Strat2b` policy it is somewhere around 60000, making `Strat2b` a more cost-efficient policy. (Indeed, the minima obtained under the two strategies over our 30 simulations were $193109/2 = 96554.5$ and $174211/3 \approx 58070.33$, respectively.) On the other hand, the long run maximum cloud costs appear to be around 170000 and 140000 under the policies `Strat3b` and `Strat2b`, respectively. (Based on our 30 simulations, they are $1749309/10 = 174930.9$ and $2876927/20 = 143846.35$, respectively.) These results allow us to conjecture that, when considering our finite state space cloud model, `Strat2b` *is a more cost-efficient policy than* `Strat3b` *in the long run.* We check this idea below by means of statistical model checking.

Remark 16. Figure 6 also shows that the *costCloud* function has a different dynamics under each policy, i.e., that, in the long run, it fluctuates more around its mean value under `Strat2b` than under `Strat3b`. This can be seen in Figure 7, which shows the sample standard deviations at each simulation step of our 30 simulated trajectories under the two policies. This could mean that, *in the long run, the total cloud costs are less predictable under the* `Strat2b` *policy than under* `Strat3b`.

To check that the policy defined by `Strat2b` is indeed more cost-efficient than `Strat3b`, we run a series of statistical model checking commands. Since statistical model checking is known

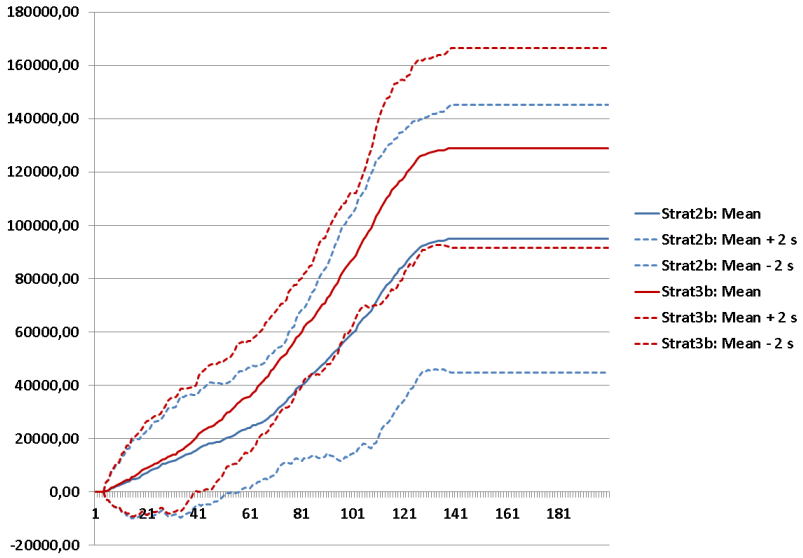

Fig. 6. Mean trajectories of the total cloud cost for the two policies quantifying our finite state space cloud model.

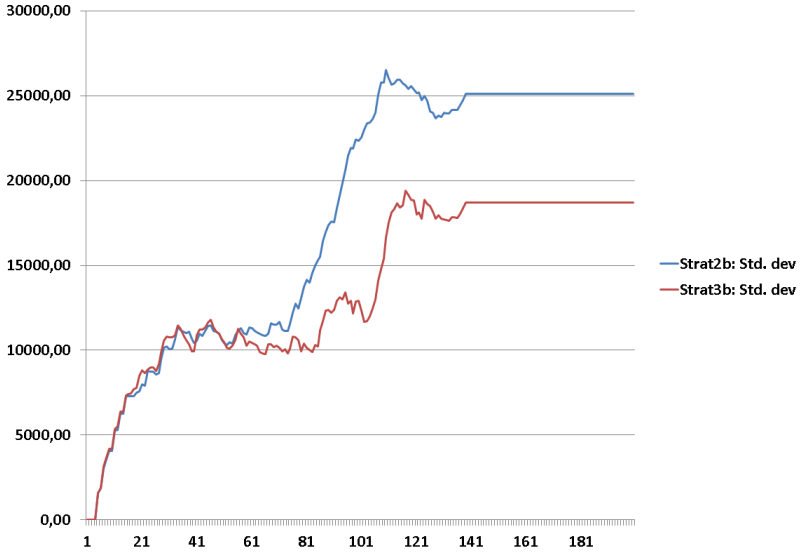

Fig. 7. Sample standard deviations at each step, for our 30 simulated trajectories under each policy.

to require large amounts of resources when running with high confidence values (very small probabilities of type I and type II errors) [32, 34], we used the approach suggested in [34] where we first verify a PCTL formula with loose error bounds, and then progressively tighten these bounds to obtain more accurate results.

We begin by defining, in a separate state predicate module `CLOUD-PRED`, a parametric state predicate `effortGreaterThan` that we use in our PCTL formulae, as follows:

```
(spmod CLOUD-PRED is protecting CLOUD-SPEC .
  smcstate Configuration . --- sort for system states
  var EFFORT : Rat .  var CF : Configuration .  var K : Rat .
  psp effortGreaterThan : Rat . --- declare the parametric state predicate
  csat (CF eff(EFFORT)) |= effortGreaterThan(K) if EFFORT > K . --- define its semantics
endspm)
```

such that `effortGreaterThan(`N`)` evaluates to `true` whenever the computational effort of the cloud service in the current state exceeds N. We then change some of the internal parameters to allow the statistical model checking to terminate in reasonable time, with slightly reduced precision. Namely, we enlarge the indifference region by increasing δ_1 to 0.05 (from 0.01), and increase the tolerance by also setting δ_2 to 0.05 (from 0.01):

```
(set delta1 0.05 .)  (set delta2 0.05 .)
```

At the same time, we decrease the stopping probability p_s from 0.1 to 0.01, to allow the generation of longer sample trajectories:

```
(set pstop 0.01 .)
```

We first use statistical model checking to more rigorously approximate the *minimum* values of the long run cloud costs under the two policies. We estimate these minima by approximating the *largest* cost level below which cost trajectories begin to fall in the long run, with small but non-negligible probability. We do this by progressively increasing the cost thresholds in our liveness PCTL formulae until the estimated probability falls below 1 by a non-negligible amount.[15]

For the `Strat2b` policy we managed to increase the level of confidence above 85%, while allowing the statistical model checking algorithm to terminate in reasonable time. We therefore run the following series of statistical model checking commands with different levels of confidence, giving the results mentioned below:[16]

- ```
  (set type1 error 0.1 .)          (set type2 error 0.1 .)
  (smc initState |= P> 0.9 [F effortGreaterThan(50000)] using Strat2b .)
  rewrites: 1718883818 in 211438626ms cpu (211469301ms real ≈ 59 h) (8129 rewrites/second)
  Result Bool: true            Number of samples used: 328905
  Confidence: 90 %             Estimated probability: 83/84 ≈ 0.98809523809
  ```

  Therefore, with an error probability below 0.1, all cloud cost trajectories eventually go beyond 50000 with a probability close to 1.

- ```
  (set type1 error 0.12 .)         (set type2 error 0.12 .)
  (smc initState |= P> 0.9 [F effortGreaterThan(60000)] using Strat2b .)
  rewrites: 2191213956 in 304523700ms cpu (304567989ms real ≈ 85 h) (7195 rewrites/second)
  Result Bool: true            Number of samples used: 297150
  Confidence: 88 %             Estimated probability: 68/71 ≈ 0.95774647887
  ```

[15] Technically, if we denote by $X_{200}^{\texttt{Strat2b}}$ and $X_{200}^{\texttt{Strat3b}}$ the random variables giving the cloud costs after 200 simulation steps under the two policies, we estimate the minima of the probability distributions followed by these random variables using small quantiles, which are known to be robust estimators.

[16] The running times are for a moderately loaded 2 GHz computer.

- `(set type1 error 0.15 .)  (set type2 error 0.15 .)`
  `(smc initState |= P> 0.9 [F effortGreaterThan(70000)] using Strat2b .)`
  `rewrites: 2276565194 in 324067700ms cpu (324114763ms real ≈ 90 h) (7024 rewrites/second)`
  `Result Bool: true  Number of samples used: 210123`
  `Confidence: 85 %  Estimated probability: 51/55 ≈ 0.92727272727`

  This last result means that, with quite high confidence, the complementary PCTL formula `P> 0.9 [G ~ effortGreaterThan(70000)]` holds in the initial state with a non-negligible probability of $1 - 51/55 \approx 0.073$, under the `Strat2b` policy. Therefore, in the long run, some cloud cost trajectories remain below this threshold of 70000.

Together with our comments in the previous section, the above results further support the idea that the minimum long run cloud cost under the `Strat2b` policy is somewhere around 60000, as previously estimated. We next run similar statistical model checking commands to approximate the minimum long run cloud cost under the `Strat3b` policy. In this case, we were able to increase the confidence to 90% (although the running times increased considerably), and therefore set the following upper bounds on the error probabilities:

`(set type1 error 0.1 .)  (set type2 error 0.1 .)`

We then ran the following series of statistical model checking commands:

- `(smc initState |= P> 0.9 [F effortGreaterThan(80000)] using Strat3b .)`
  `rewrites: 1756949408 in 208854136ms cpu (208894039ms real ≈ 58 h) (8412 rewrites/second)`
  `Result Bool: true  Number of samples used: 342000`
  `Confidence: 90 %  Estimated probability: 1`
- `(smc initState |= P> 0.9 [F effortGreaterThan(90000)] using Strat3b .)`
  `rewrites: 2262645367 in 323433600ms cpu (323488576ms real ≈ 90 h) (6995 rewrites/second)`
  `Result Bool: true  Number of samples used: 367110`
  `Confidence: 90 %  Estimated probability: 83/84 ≈ 0.98809523809`

  This result means that, with a high confidence of 90%, the complementary PCTL formula `P> 0.9 [G ~ effortGreaterThan(90000)]` holds in the initial state with a rather small probability of $1 - 83/84 \approx 0.012$, under the `Strat3b` policy. Therefore, in the long run, a few cloud cost trajectories remain below 90000, but more than 98% exceed this threshold.
- `(smc initState |= P> 0.9 [F effortGreaterThan(100000)] using Strat3b .)`
  `rewrites: 3548539842 in 743127530ms cpu (743245497ms real ≈ 206 h) (4775 rewrites/second)`
  `Result Bool: false  Number of samples used: 391680`
  `Confidence: 90 %  Estimated probability: 25/28 ≈ 0.89285714285`

  This last result means that, with high confidence, the complementary PCTL formula `P> 0.9 [G ~ effortGreaterThan(100000)]` holds in the initial state with a non-negligible probability of $1 - 25/28 \approx 0.1$, under the `Strat3b` policy. Therefore, in the long run, some cloud cost trajectories remain below this threshold of 100000.

To conclude, together with our comments in the previous section, the above results further support the idea that the minimum long run cloud cost under the `Strat3b` policy is somewhere around 100000, which is higher than the estimated minimum under the `Strat2b` policy. This supports the idea that the latter *can* be a more cost-efficient policy than `Strat3b`, up to the given confidence values and up to the approximation errors for estimating minima.

Finally, we compare the maxima of the long run cloud costs under the two policies. We first check, with a smaller confidence of 73% (but using the same internal parameters as for the previous commands) that, with high probability, the cloud costs always remain below the thresholds of 140000 and 160000 under the `Strat2b` and `Strat3b` policies, respectively. We therefore set the following error probabilities:

```
(set type1 error 0.27 .)          (set type2 error 0.27 .)
```

and first check that the cloud cost never exceeds 140000 under the `Strat2b` policy:

```
(smc initState |= P> 0.9 [G ~ effortGreaterThan(140000)] using Strat2b .)
rewrites: 1558476059 in 191669025ms cpu (191710196ms real ≈ 53 h) (8131 rewrites/second)
Result Bool: true            Number of samples used: 7072
Confidence: 73 %             Estimated probability: 1
```

Similarly, we check that the cloud cost never exceeds 160000 under the `Strat3b` policy:

```
(smc initState |= P> 0.9 [G ~ effortGreaterThan(160000)] using Strat3b .)
rewrites: 2038529739 in 285936192ms cpu (285995805ms real ≈ 80 h) (7129 rewrites/second)
Result Bool: true            Number of samples used: 13052
Confidence: 73 %             Estimated probability: 19/20 = 0.95
```

With an error probability below 0.27, these results support the idea that the maximum under the `Strat2b` policy is below 140000, whereas the maximum under the `Strat3b` policy is somewhere around 160000. It therefore seems that, even in the worst case, `Strat2b` is still more cost-efficient than `Strat3b`. However, further analysis is necessary, using smaller error probabilities.

Remark 17. The results obtained in this section differ slightly than the ones in [6], but the conclusion, also reached by means of statistical model checking in this report, is the same: *in the long run,* `Strat2b` *is more cost-efficient than* `Strat3b`. As we also explained in Section 5, the reason is that in [6] we used a prototype version of our tool that had a bug in the pseudo-random number generation, which affected the simulation and statistical model checking results. Furthermore, the probabilistic strategies that we used in that paper to formalize the two policies were ill-defined, something that we only noticed after we further developed the theory in this technical report. For these reasons, the statistical model checking results reported in [6] for the cloud computing example are incorrect, but are corrected in this report.

7 Related Work

A number of tools support models that are both probabilistic and nondeterministic, including Markov automata [16], generalized stochastic Petri nets [25], or uniform labeled transition systems [7].

In probabilistic automaton-based models one can use synchronous parallel composition to quantify nondeterministic choices by composing the system with a new "scheduler" component. For example, if the model allows us to nondeterministically select action a or action b, we can quantify this nondeterminism by composing the model with a "scheduler" automaton with transitions $\{s_0 \xrightarrow{\tau\,[1/3]} \textit{do-}a,\ s_0 \xrightarrow{\tau\,[2/3]} \textit{do-}b,\ \textit{do-}a \xrightarrow{\overline{a}} s_0,\ \textit{do-}b \xrightarrow{\overline{b}} s_0\}$; the composed system will then do action a with probability $1/3$ and action b with probability $2/3$. Such "scheduler" components are supported by tools like MODEST [8] and PRISM [22]. Our approach contrasts with this one by: (*i*) having a more explicit separation between model and strategy, since in the above approach the strategy is just another "system" component; (*ii*) supporting a more expressive underlying specification language with unbounded data types, dynamic object/message creation and deletion, and arbitrary complex data types; and (*iii*) providing a more expressive and convenient way of specifying the strategies themselves. It is also unclear to what extent automaton-based approaches can support *hierarchical* systems.

In Uppaal-SMC [14] the nondeterminism concerning *which* component should be executed next is *implicitly* resolved by assigning a stochastic delay to each component; the one with the

shortest remaining delay is then scheduled for execution. If multiple transitions are enabled for a component, then one is chosen uniformly at random. In contrast, our language allows the user to specify the probability distributions that quantify the nondeterminism in the model.

Maude itself has a non-probabilistic strategy language [17] to guide the execution of *non-probabilistic* rewrite theories; i.e., there is no support for quantifying the nondeterminism either in the model or in the strategy. VeStA [33] can analyze fully probabilistic actor PMaude specifications. For *flat* object-oriented systems nondeterminism is typically removed by letting each action be triggered by a message, and letting probabilistic rules add a stochastic delay to each message [1, 3]. The probability of two messages being scheduled at the same "time" is then zero, and this therefore resolves nondeterminism. This method was recently extended to *hierarchical* object-oriented systems [15, 11]. The differences with our work are: (*i*) whereas we have a clear separation between system model and adversary, the above approach encodes the adversary in the model, and hence clutters it with fictitious clocks and schedulers to obtain a fully probabilistic model; (*ii*) our implementation supports not only a subset of object-oriented specifications, but the entire class of PRTs with fixed-size probability distributions; and (*iii*) we add a simulator and statistical model checker to Maude instead of using an external tool.

ELAN [9] is a rewriting language where strategies are first class citizens that can appear in rewrite rules, so there is no separation between system model and strategies. The paper [10] adds a "probabilistic choice" operator $\texttt{PC}(s_1 : p_1, \ldots, s_n : p_n)$ to ELAN's strategy language, where p_i defines the probability of applying strategy s_i. This approach is different from ours in the following ways: there is no separation between "system" and "strategy;" the definition of context and substitution adversaries is not supported and therefore not all nondeterminism in a system can be quantified; and there is no support for probabilistic model checking analysis.

8 Conclusions

In this paper we define what is, to the best of our knowledge, the first language for defining complex probabilistic strategies to quantify the nondeterminism in infinite-state systems with both probabilities and nondeterminism, and that includes the object-oriented systems with dynamic object creation. We propose a modular safety/QoS analysis methodology in which a simple (typically non-probabilistic) "base" model that defines all possible behaviors of the system can be easily verified for safety, and where different probabilistic refinements can be specified in our language on top of this verified base model. QoS properties of the different refinements can then be analyzed by statistical and exact probabilistic model checking.

We have implemented a probabilistic simulator and statistical PCTL model checker for our strategies for all probabilistic rewrite theories with discrete probability distributions. We have shown the usefulness of our language and methodology using a cloud computing example, where different probabilistic strategies on top of the verified base model define different load balancing policies for the cloud, and have shown how our tool PSMaude can be used to compare the QoS provided by different policies. This example indicates that we need to integrate timed modeling and analysis into our framework.

We also plan to extend PSMaude to allow the statistical analysis of quantitative temporal expressions specified in the QuaTEx logic [1]. This would allow, e.g., the statistical estimation of *expected values* of particular numerical quantities associated with a probabilistic rewrite theory whose nondeterminism is quantified by a given probabilistic strategy. Another direction for future work is to investigate algorithms for the *exact* (vs. statistical) probabilistic model checking of probabilistic rewrite theories for which *some*, but not necessarily all nonde-

terminism is quantified by a "partial" probabilistic strategy. Furthermore, since our proposed probabilistic strategy language only allows defining memoryless adversaries, we also aim to extend its syntax and semantics to allow defining *history-dependent* adversaries.

Finally, since our tool is a prototype and due to the generality of our strategy language, the performance of our tool could further be improved.

Acknowledgments. We thank José Meseguer and Roy Campbell for discussions on probabilistic strategies, and gratefully acknowledge partial support for this work by AFOSR Grant FA8750-11-2-0084.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. ENTCS 153(2) (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO. LNCS, vol. 6859. Springer (2011)
3. AlTurki, M., Meseguer, J., Gunter, C.A.: Probabilistic modeling and analysis of DoS protection for the ASV protocol. ENTCS 234 (2009)
4. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time Markov chains. In: Alur, R., Henzinger, T.A. (eds.) CAV. LNCS, vol. 1102, pp. 269–276. Springer (1996)
5. Bentea, L.: The PSMaude tool home page: `http://www.ifi.uio.no/~lucianb/psmaude/`
6. Bentea, L., Ölveczky, P.C.: A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In: Martí-Oliet, N., Palomino, M. (eds.) WADT. LNCS, vol. 7841, pp. 77–94. Springer (2012)
7. Bernardo, M., Nicola, R., Loreti, M.: Uniform labeled transition systems for nondeterministic, probabilistic, and stochastic processes. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) Trustworthly Global Computing, LNCS, vol. 6084, pp. 35–56. Springer Berlin Heidelberg (2010)
8. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MODEST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. 32(10), 812–830 (2006)
9. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Ringeissen, C.: An overview of ELAN. Electronic Notes in Theoretical Computer Science 15 (1998)
10. Bournez, O., Kirchner, C.: Probabilistic rewrite strategies. Applications to ELAN. In: RTA. LNCS, vol. 2378. Springer (2002)
11. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with Maude. In: Durán, F. (ed.) WRLA. LNCS, vol. 7571, pp. 118–138. Springer (2012)
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2) (1986)
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
14. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: CAV. LNCS, vol. 6806. Springer (2011)
15. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: FASE. LNCS, vol. 7212. Springer (2012)
16. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS. pp. 342–351. IEEE Computer Society (2010)
17. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. Electronic Notes in Theoretical Computer Science 174(11), 3–25 (2007)
18. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6 (1994)

19. Jansen, D.N., Katoen, J.P., Oldenkamp, M., Stoelinga, M., Zapreev, I.S.: How fast and fat is your probabilistic model checker? an experimental performance comparison. In: Yorav, K. (ed.) Haifa Verification Conference. LNCS, vol. 4899, pp. 69–85. Springer (2007)
20. Katelman, M., Meseguer, J., Hou, J.C.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: FMOODS. LNCS, vol. 5051. Springer (2008)
21. Kumar, N., Sen, K., Meseguer, J., Agha, G.: Probabilistic rewrite theories: Unifying models, logics and tools. Technical report UIUCDCS-R-2003-2347, Department of Computer Science, University of Illinois at Urbana-Champaign (2003)
22. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806. Springer (2011)
23. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
24. López, G.G.I., Hermanns, H., Katoen, J.P.: Beyond memoryless distributions: Model checking semi-Markov chains. In: PAPM-PROBMIV. LNCS, vol. 2165. Springer (2001)
25. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic Petri nets. SIGMETRICS Performance Evaluation Review 26(2), 2 (1998)
26. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science 285(2) (2002)
27. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for maude. Electronic Notes in Theoretical Computer Science 117, 417–441 (2005)
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1) (1992)
29. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: WADT'97, LNCS, vol. 1376. Springer (1997)
30. Meseguer, J.: A rewriting logic sampler. In: ICTAC. LNCS, vol. 3722. Springer (2005)
31. Saw, J.G., Yang, M.C.K., Mo, T.C.: Chebyshev inequality with estimated mean and variance. The American Statistician 38(2), 130–132 (1984)
32. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV. LNCS, vol. 3576. Springer (2005)
33. Sen, K., Viswanathan, M., Agha, G.A.: VeStA: A statistical model-checker and analyzer for probabilistic systems. In: QEST'05. IEEE Computer Society (2005)
34. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. LNCS, vol. 2404. Springer (2002)

A Computing the Exact Probabilities in the Blackboard Example

Below we give a parallel Mathematica code for computing the exact probabilities in the blackboard game specification in Section 5.

```
exactProb[state_, bound_] := 1 - exactComplementaryProbParallelFirst[state, bound]

exactComplementaryProbParallelFirst[state_, bound_] :=
 If[Total[state] > bound, 1,
  If[Length[state] == 1, 0,
   Block[{total = 0, i, j, sum = 0, min, max, common, nextStateMin2,
     nextStateMin3, nextStateMax2, nextStateMax3},

    (* compute (inverse of) normalization factor *)
    For[i = 1, i < Length[state], i++,
     For[j = i + 1, j <= Length[state], j++,
      sum += 1/(state[[i]]*state[[j]])
      ]
     ];

    For[i = 1, i < Length[state], i++,
     For[j = i + 1, j <= Length[state], j++,
      (* pick the context corresponding to the index pair (i, j) *)
      min = Min[state[[i]], state[[j]]];
      max = Max[state[[i]], state[[j]]];
      erasePair = Delete[state, {{i}, {j}}];

      (* pick a substitution for LHS and one for the probabilistic RHS variables *)
      (* also compute the probabilities of the 4 successor states for the index pair (i, j) *)
      nextStateMin2 = Append[erasePair, Floor[(min^2 + max)/2]];
      nextStateMin3 = Append[erasePair, Floor[(min^3 + max)/2]];
      nextStateMax2 = Append[erasePair, Floor[(max^2 + min)/2]];
      nextStateMax3 = Append[erasePair, Floor[(max^3 + min)/2]];

      (* sum up the probability over all successor states *)
      common = 1/(min*max*sum*40);
      total +=
       common *
        Dot[ParallelMap[
          exactComplementaryProb[#, bound] &, {nextStateMin2,
           nextStateMin3, nextStateMax2, nextStateMax3}], {27, 9, 3, 1}]
      ]
     ];
    total
    ]
   ]
  ]
```

```
exactComplementaryProb[state_, bound_] :=
 If[Total[state] > bound, 1,
  If[Length[state] == 1, 0,
   Block[{total = 0, i, j, sum = 0, common, nextStateMin2,
     nextStateMin3, nextStateMax2, nextStateMax3},

    (* compute (inverse of) normalization factor *)
    For[i = 1, i < Length[state], i++,
     For[j = i + 1, j <= Length[state], j++,
      sum += 1/(state[[i]]*state[[j]])
      ]
     ];

    For[i = 1, i < Length[state], i++,
     For[j = i + 1, j <= Length[state], j++,
      (* pick the context corresponding to the index pair (i, j) *)
      min = Min[state[[i]], state[[j]]];
      max = Max[state[[i]], state[[j]]];
      erasePair = Delete[state, {{i}, {j}}];

      (* pick a substitution for LHS and one for the probabilistic RHS variables *)
      (* also compute the probabilities of the 4 successor states for the index pair (i, j) *)
      nextStateMin2 = Append[erasePair, Floor[(min^2 + max)/2]];
      nextStateMin3 = Append[erasePair, Floor[(min^3 + max)/2]];
      nextStateMax2 = Append[erasePair, Floor[(max^2 + min)/2]];
      nextStateMax3 = Append[erasePair, Floor[(max^3 + min)/2]];

      (* sum up the probability over all successor states *)
      common = 1/(min*max*sum*40);
      total +=
       common *
        Dot[Map[exactComplementaryProb[#, bound] &, {nextStateMin2,
           nextStateMin3, nextStateMax2, nextStateMax3}], {27, 9, 3, 1}]
      ]
     ];
    total
    ]
   ]
  ]

exactProb[{2, 3, 5, 7, 11, 13, 17}, 1000000]
```

B Full Specification of the Unbounded Cloud Computing Model

We provide the full PSMaude specification of the infinite state space cloud computing model presented in Section 6.

```
(pmod CLOUD-SPEC is protecting RAT .
  vars A B D M M' N N' T NEXTID VMNOREQ VMLD VMLD' VMLDR VMFL VMFL' VMNO : Nat .
  var R : String .
  vars LOC : Location .
  vars AS1 AS2 : CServiceDataSet .
  vars DC PS VM VM' : Configuration .
  vars O O' CSO DCO PSO PSO' VMO : Oid .
  var OIDSET : OidMSet .

  ---------------------------------- STATIC PART ----------------------------------------------

  class Node | priority : Nat .  --- the priority for resolving requests from this node
  class SUser | location : Location .  --- location of this user
  class SProvider .
  subclasses SUser SProvider < Node .  --- class inheritance
  --- the object ID is a node's IP address
  op ip : Nat -> Oid .

  class Region | location : Location,           --- the location of the region
                 dataCenters : Configuration,   --- the data centers in a region
                 vmNo : Nat,                    --- the total number of VMs in this region
                 vmLoad : Nat .                 --- the total workload on all VMs in this region
  --- each region has a name given by a string
  subsort String < Oid .

  class DCenter | pservers : Configuration .  --- the physical servers in a data center

  class PServer | load : Nat,           --- the number of VMs running on a physical server
                  maxLoad : Nat,        --- the maximum number of VMs allowed
                  nextVMID : Nat,       --- counter for generating fresh IDs for new VMs
                  vmLoad : Nat,         --- the total workload of all VMs on the server
                  vmFailed : Nat,       --- the number of failed VMs on the server
                  vms : Configuration . --- the virtual machines running on the server

  --- computes the failure ratio of a server with total load M and number of VMs that failed VMFL
  op failureRatio : Nat Nat -> Rat .
  eq failureRatio(VMFL, M) = if M == 0 then 0 else VMFL / M fi .

  class VMachine | owner : Oid,        --- the object ID of the owner of this virtual machine
                   vmReq : OidMSet,    --- IDs of users whose requests are resolved on the VM
                   vmMaxReq : Nat,     --- maximum number of requests resolved simultaneously
                   running : Bool,     --- whether the VM is running (false if it crashed)
                   throughput : Nat .  --- number of requests resolved so far on the VM
  --- unique identification pair of a virtual machine
  op vm : Oid Nat -> Oid .

  --- multiset of object IDs
  sort OidMSet .  subsort Oid < OidMSet .
  op noid : -> OidMSet [ctor] .
  op __ : OidMSet OidMSet -> OidMSet [ctor assoc comm id: noid] .
  --- cardinality of such a multiset
  op size : OidMSet -> Nat [memo] .
  eq size(noid) = 0 .  eq size(O OIDSET) = 1 + size(OIDSET) .
```

```
--- sorts for data stored on the cloud service
sorts CServiceData CServiceDataSet .
subsort CServiceData < CServiceDataSet .
op _‘,_ : CServiceDataSet CServiceDataSet -> CServiceDataSet [ctor assoc comm] .

class CService | status : CServiceDataSet,  --- status data stored by the cloud service
                 subscr : CServiceDataSet . --- subscription data
---  noReq(ID, K) = current number of requests of a SUser
---   noVM(ID, K) = current number of VMs of a SProvider
--- maxReq(ID, N) = maximum no. requests running for a SUser
---  maxVM(ID, N) = maximum no. VMs running for a SProvider
ops noReq maxReq noVM maxVM : Oid Nat -> CServiceData .

---------------------------------- MESSAGES ----------------------------------------------

--- user request message
--- req(A, B) is a request from SUser A sent to the web application of SProvider B
msg req : Oid Oid -> Msg .

--- provider request message
--- e.g., launch(A, "Europe") is a request from SProvider A
--- to launch a new VM in the region "Europe"
msg launch : Oid String -> Msg .

-------------------------- DISTANCES BETWEEN LOCATIONS -----------------------------------

--- some locations
sort Location .
ops locUS locEU locA locB locC locD : -> Location [ctor] .

--- the distance between locations in kilometers
op distance : Location Location -> Nat [comm memo] .
eq distance(LOC, LOC) = 0 .

--- approximate distance between US and EU
eq distance(locUS, locEU) = 6000 .

--- distances from US to the users
eq distance(locUS, locA) = 750 .   eq distance(locUS, locB) = 375 .
eq distance(locUS, locC) = 750 .   eq distance(locUS, locD) = 6047 .

--- distances from EU to the users
eq distance(locEU, locA) = 6047 .  eq distance(locEU, locB) = 6375 .
eq distance(locEU, locC) = 6047 .  eq distance(locEU, locD) = 750 .

--- distances among the users close to US
eq distance(locA, locB) = 839 .    eq distance(locA, locC) = 1500 .
eq distance(locB, locC) = 839 .

--- distances from the user close to EU, to the users close to US
eq distance(locA, locD) = 6000 .   eq distance(locB, locD) = 6419 .
eq distance(locC, locD) = 6185 .
```

```
---------------------------------- INITIAL STATE ----------------------------------------

--- some physical servers of different capacities
ops psUS1 psUS2 psEU1 psEU2 : -> Configuration .
eq psUS1 = < ip(4000) : PServer | load : 0, maxLoad : 100, nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none >
           < ip(4001) : PServer | load : 0, maxLoad : 50,  nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none > .
eq psUS2 = < ip(4002) : PServer | load : 0, maxLoad : 100, nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none >
           < ip(4003) : PServer | load : 0, maxLoad : 50,  nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none > .
eq psEU1 = < ip(5000) : PServer | load : 0, maxLoad : 200, nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none >
           < ip(5001) : PServer | load : 0, maxLoad : 40,  nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none > .
eq psEU2 = < ip(5002) : PServer | load : 0, maxLoad : 200, nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none >
           < ip(5003) : PServer | load : 0, maxLoad : 40,  nextVMID : 0,
                                  vmLoad : 0, vmFailed : 0, vms : none > .
--- some data centers in each region
ops dcUS dcEU : -> Configuration .
eq dcUS = < ip(400) : DCenter | pservers : psUS1 >  < ip(401) : DCenter | pservers : psUS2 > .
eq dcEU = < ip(500) : DCenter | pservers : psEU1 >  < ip(501) : DCenter | pservers : psEU2 > .


--- a possible initial state of the cloud system
op initState : -> Configuration .
eq initState = < ip(100) : SUser | location : locA, priority : 1 >
               < ip(101) : SUser | location : locB, priority : 3 >
               < ip(102) : SUser | location : locC, priority : 3 >
               < ip(103) : SUser | location : locD, priority : 5 >
               < ip(200) : SProvider | priority : 2 >
               < ip(201) : SProvider | priority : 5 >
               < ip(300) : CService | status : ( noReq(ip(100), 0),   noReq(ip(101), 0),
                                                 noReq(ip(102), 0),   noReq(ip(103), 0),
                                                  noVM(ip(200), 0),    noVM(ip(201), 0)),
                                      subscr : (maxReq(ip(100), 20), maxReq(ip(101), 15),
                                                maxReq(ip(102), 50), maxReq(ip(103), 50),
                                                 maxVM(ip(200), 10),  maxVM(ip(201), 10)) >
               <     "US" : Region | location : locUS, dataCenters : dcUS, vmNo : 0, vmLoad : 0 >
               < "Europe" : Region | location : locEU, dataCenters : dcEU, vmNo : 0, vmLoad : 0 >
               launch(ip(200), "US")  launch(ip(200), "Europe")
               launch(ip(201), "US")  launch(ip(201), "Europe") .



---------------------------------- MODEL PARAMETERS ------------------------------------

--- maximum number of requests that a VM can resolve simultaneously (the same for all VMs)
op MAXREQ : -> Nat .  eq MAXREQ = 3 .
```

```
---------------------------------- DYNAMIC PART -------------------------------------------

--- Distribute a request from a SUser O to an application that the SProvider O' is running,
--- such that its new number of requests (A + 1) does not exceed the max. subscription value B.
--- The system picks a region, a data center, a server and a running VM of SProvider O'
--- such that it can still resolve a new request (size(OIDSET) < D).
crl [processUserReq]:
    req(O, O')  < CSO : CService | status : (noReq(O, A), AS1), subscr : (maxReq(O, B), AS2) >
    < R : Region |      vmLoad : VMLDR,
                  dataCenters : < DCO : DCenter | pservers : < PSO : PServer |
                                                               vmLoad : VMLD,
                                                                  vms : < VMO : VMachine |
                                                                              owner : O',
                                                                            running : true,
                                                                              vmReq : OIDSET,
                                                                            vmMaxReq : D
                                                                          > VM
                                                             > PS
                                > DC
    >
 => < CSO : CService | status : (noReq(O, A + 1), AS1) >
    < R : Region |      vmLoad : VMLDR + 1,
                  dataCenters : < DCO : DCenter | pservers : < PSO : PServer |
                                                               vmLoad : VMLD + 1,
                                                                  vms : < VMO : VMachine |
                                                                              vmReq : OIDSET O
                                                                          > VM
                                                             > PS
                                > DC
    >
if A < B /\ size(OIDSET) < D .

--- Resolve an SUser request req(O, O') on the virtual machine of some provider O'
--- selected in rule [processUserReq] and increase the throughput attribute of the selected VM.
rl [resolveUserReq]:
   < CSO : CService | status : (noReq(O, A), AS1) >
   < R : Region |      vmLoad : VMLDR,
                 dataCenters : < DCO : DCenter | pservers : < PSO : PServer  |
                                                              vmLoad : VMLD,
                                                                 vms : < VMO : VMachine |
                                                                             running : true,
                                                                               vmReq : O OIDSET,
                                                                           throughput : T
                                                                         > VM
                                                            > PS
                               > DC
   >
=> < CSO : CService | status : (noReq(O, A - 1), AS1) >
   < R : Region |      vmLoad : VMLDR - 1,
                 dataCenters : < DCO : DCenter | pservers : < PSO : PServer  |
                                                              vmLoad : VMLD - 1,
                                                                 vms : < VMO : VMachine |
                                                                             vmReq : OIDSET,
                                                                           throughput : T + 1
                                                                         > VM
                                                            > PS
                               > DC
   > .
```

```
--- Process a provider request from a SProvider O and launch a new VM in the region R.
--- The system picks one of the dataCenters nondeterministically,
--- and then picks one of the physical servers nondeterministically
--- such that the server's capacity is not exceeded by running a new VM,
--- and such that the subscription of the SProvider O allows running another VM.
crl [processProviderReq]:
    launch(O, R)
    < CSO : CService | status : (noVM(O, A), AS1), subscr : (maxVM(O, B), AS2) >
    < R : Region |        vmNo : VMNO,
                   dataCenters : < DCO : DCenter |
                                   pservers : < PSO : PServer |
                                                   load : M,
                                                maxLoad : N,
                                              nextVMID : NEXTID,
                                                   vms : VM
                                              > PS
                                 > DC
    >
 => < CSO : CService | status : (noVM(O, A + 1), AS1) >
    < R : Region |        vmNo : VMNO + 1,
                   dataCenters : < DCO : DCenter |
                                   pservers : < PSO : PServer |
                                                   load : M + 1,
                                              nextVMID : NEXTID + 1,
                                                   vms : VM
                                                         < vm(PSO, NEXTID) : VMachine |
                                                                owner : O,
                                                             running : true,
                                                               vmReq : noid,
                                                            vmMaxReq : MAXREQ,
                                                          throughput : 0
                                                         >
                                              > PS
                                 > DC
    >
if A < B /\ M < N .



--- One of the VMs may crash at any time.
rl [failVM]:
   < PSO : PServer | vmFailed : VMFL,
                     vms : < VMO : VMachine | running : true >  VM
   >
=> < PSO : PServer | vmFailed : VMFL + 1,
                     vms : < VMO : VMachine | running : false > VM
   > .
```

```
  --- Within one of the data centers of one of the regions (both chosen nondeterministically),
  --- the system migrates a running VM from a PServer PSO to a PServer PSO'.
  --- This event can only take place if the two servers have different failure ratios,
  --- (i.e., the system does not migrate VMs between servers with the same failure ratio),
  --- and provided that the "source" server is unreliable, i.e., it has at least one failed VM.
  --- Furthermore, the "destination" server should have enough resources to store the migrated VM.
  crl [migrateVM]:
      < PSO  : PServer |     load : M,
                         vmLoad : VMLD,
                       vmFailed : VMFL,
                            vms : VM < VMO : VMachine | vmReq : OIDSET, running : true >
      >
      < PSO' : PServer |     load : M',
                         vmLoad : VMLD',
                        maxLoad : N',
                       vmFailed : VMFL',
                            vms : VM'
      >
   => < PSO  : PServer |     load : M - 1,
                         vmLoad : VMLD - VMNOREQ,
                            vms : VM
      >
      < PSO' : PServer |     load : M' + 1,
                         vmLoad : VMLD' + VMNOREQ,
                            vms : VM' < VMO : VMachine | >
      >
  if M' < N' /\ VMFL > 0 /\ failureRatio(VMFL, M) =/= failureRatio(VMFL', M')
  /\ VMNOREQ := size(OIDSET) .



  --- A new user request is sent from some SUser O to a web application
  --- that a SProvider O' is running.
  rl [newUserReq]:
     < O : SUser | >  < O' : SProvider | >
  => < O : SUser | >  < O' : SProvider | >  req(O, O') .


  --- A new provider request is sent from a SProvider O to one of the regions,
  --- both selected nondeterministically.
  rl [newProviderReq]:
     < O : SProvider | >  < R : Region | >
  => < O : SProvider | >  < R : Region | >  launch(O, R) .

endpm)
```

C Full Specification of the `Strat2b` Probabilistic Policy for the Unbounded Cloud Computing Model

We give below the full PSMaude specification of the `Strat2b` probabilistic strategy, quantifying all nondeterminism in the infinite state space cloud computing model presented in Section 6.

```
(psmod CLOUD-STRAT2B is protecting CLOUD-SPEC .
  state Configuration . --- sort for system states

  --- Rule strategies model the probabilities of high-level decisions in the system.
  psdrule RuleStrat := given state: CF:Configuration
                               is: ( resolveUserReq )     -> 10 ;
                                   ( processUserReq )     -> 1000 ;
                                   ( processProviderReq ) -> 100 ;
                                   ( newUserReq )         -> 100 ;
                                   ( newProviderReq )     -> 10 ;
                                   ( failVM )             -> 1 ;
                                   ( migrateVM )          -> 1
  [none] .

  --- Context strategies model the probabilities of
  --- generating, processing and resolving different messages in the system

  --- User requests with high priorities are selected with high probability
  --- (the quadratic factor P * P) together with a region R,
  --- where the probability is also inversely proportional to the distance
  --- distance(LOC, LOC') between the user and the region R.
  psdcontext CtxStrat :=
    given state: CF:Configuration
                 < O:Oid : SUser | priority : P:Nat, location : LOC:Location >
                 req(O:Oid, O':Oid)
                 < CSO:Oid : CService | ATTS:AttributeSet >
                 < R:String : Region | location : LOC':Location, ATTS':AttributeSet >
           rule: processUserReq
             is: (CF:Configuration < O:Oid : SUser | priority : P:Nat, location : LOC:Location > [])
                 -> ((P:Nat * P:Nat) / (1 + distance(LOC:Location, LOC':Location)))
  [none] .

  --- Model how the load balancer selects with high probability a VM with small workload
  --- size(XOIDSET) to forward the user request to. Using an implicit factor of 1
  --- in the weight expression below, we also model how the data center and the server
  --- matched by the processUserReq rule are picked uniformly at random.
  psdsubst SubstStrat :=
    given state: CF:Configuration
                 req(XO:Oid, XO':Oid)
                 < XCSO:Oid : CService | status :  (noReq(XO:Oid, XA:Nat), XAS1:CServiceDataSet),
                                         subscr : (maxReq(XO:Oid, XB:Nat), XAS2:CServiceDataSet) >
                 < XR:String : Region | ATTS:AttributeSet,
                   vmLoad : XVMLDR:Nat,
                   dataCenters : < XDCO:Oid : DCenter |
                                   pservers : < XPSO:Oid : PServer | ATTS':AttributeSet,
                                                vmLoad : XVMLD:Nat,
                                                   vms : < XVMO:Oid : VMachine | ATTS'':AttributeSet,
                                                               owner : XO':Oid,
                                                             running : true,
                                                               vmReq : XOIDSET:OidMSet,
                                                            vmMaxReq : XD:Nat
                                                          >
```

```
                                                    XVM:Configuration
                                                >
                                                XPS:Configuration
                                >
                                XDC:Configuration
                >
           rule: processUserReq
        context: CF:Configuration []
             is: { O:Oid <- XO:Oid, O':Oid <- XO':Oid, CSO:Oid <- XCSO:Oid,
                   A:Nat <- XA:Nat, AS1:CServiceDataSet <- XAS1:CServiceDataSet,
                   B:Nat <- XB:Nat, AS2:CServiceDataSet <- XAS2:CServiceDataSet,
                   R:String <- XR:String, VMLDR:Nat <- XVMLDR:Nat,
                   DCO:Oid <- XDCO:Oid, PSO:Oid <- XPSO:Oid, VMLD:Nat <- XVMLD:Nat,
                   VMO:Oid <- XVMO:Oid, OIDSET:OidMSet <- XOIDSET:OidMSet, D:Nat <- XD:Nat,
                   VM:Configuration <- XVM:Configuration, PS:Configuration <- XPS:Configuration,
                   DC:Configuration <- XDC:Configuration }
                 -> (1 / (1 + size(XOIDSET:OidMSet)))
[none] .


--- One of the user requests that are currently being processed on one of the VMs is resolved,
--- both choices being uniform.
psdcontext CtxStrat := given state: CF:Configuration
                              rule: resolveUserReq
                                is: uniform
[none] .

psdsubst SubstStrat := given state: CF:Configuration
                              rule: resolveUserReq
                           context: CTX:Configuration
                                is: uniform
[none] .


--- Model the fact that requests from providers with high priorities
--- are selected with higher probability (the factor P * P in the weight expression below).
psdcontext CtxStrat := given state: CF:Configuration
                                    < O:Oid : SProvider | priority : P:Nat >
                                    launch(O:Oid, R:String)
                                    < CSO:Oid : CService | ATTS:AttributeSet >
                                    < R:String : Region | ATTS':AttributeSet >
                              rule: processProviderReq
                                is: (CF:Configuration < O:Oid : SProvider | priority : P:Nat > [])
                                    -> (P:Nat * P:Nat)
[none] .
```

```
--- Model how, with high probability, the new VM is launched on a physical server
--- with small load XM within the given region XR.
--- The data center object is also (implicitly) selected uniformly at random.
psdsubst SubstStrat :=
  given state: CF:Configuration
               launch(XO:Oid, XR:String)
               < XCSO:Oid : CService | status :  (noVM(XO:Oid, XA:Nat), XAS1:CServiceDataSet),
                                       subscr : (maxVM(XO:Oid, XB:Nat), XAS2:CServiceDataSet) >
               < XR:String : Region | ATTS:AttributeSet,
                        vmNo : XVMNO:Nat,
                 dataCenters : < XDCO:Oid : DCenter |
                                 pservers : < XPSO:Oid : PServer | ATTS':AttributeSet,
                                              load : XM:Nat,
                                              maxLoad : XN:Nat,
                                              nextVMID : XNEXTID:Nat,
                                              vms : XVM:Configuration
                                            >
                                            XPS:Configuration
                               >
                               XDC:Configuration
               >
         rule: processProviderReq
      context: CF:Configuration []
           is: { O:Oid <- XO:Oid, R:String <- XR:String, CSO:Oid <- XCSO:Oid,
                 A:Nat <- XA:Nat, AS1:CServiceDataSet <- XAS1:CServiceDataSet,
                 B:Nat <- XB:Nat, AS2:CServiceDataSet <- XAS2:CServiceDataSet,
                 VMNO:Nat <- XVMNO:Nat, DCO:Oid <- XDCO:Oid, PSO:Oid <- XPSO:Oid,
                 M:Nat <- XM:Nat, N:Nat <- XN:Nat, NEXTID:Nat <- XNEXTID:Nat,
                 VM:Configuration <- XVM:Configuration, PS:Configuration <- XPS:Configuration,
                 DC:Configuration <- XDC:Configuration }
               -> (1 / (1 + XM:Nat))
[none] .


--- Model the fact that service users are 5 times more likely to send requests
--- to the application of the service provider with IP 201,
--- than to the application of the service provider with IP 200.
psdcontext CtxStrat :=
  given state: CF:Configuration
               < ip(N:Nat) : SUser | ATTS:AttributeSet >
               < ip(200) : SProvider | ATTS':AttributeSet >
               < ip(201) : SProvider | ATTS'':AttributeSet >
         rule: newUserReq
           is: --- generate  req(ip(N), 200)
               (CF:Configuration < ip(201) : SProvider | ATTS'':AttributeSet > []) -> 1 ;
               --- generate  req(ip(N), 201)
               (CF:Configuration < ip(200) : SProvider | ATTS':AttributeSet >  []) -> 5
[none] .

--- Uniform substitution strategy, since there is always a single matching substitution.
psdsubst SubstStrat := given state: CF:Configuration
                              rule: newUserReq
                           context: CTX:Configuration
                                is: uniform
[none] .
```

```
--- One of the service providers generates a new "launch VM" request
--- to one of the regions - both selected uniformly at random.
psdcontext CtxStrat := given state: CF:Configuration
                             rule: newProviderReq
                               is: uniform
[none] .

psdsubst SubstStrat := given state: CF:Configuration
                             rule: newProviderReq
                          context: CTX:Configuration
                               is: uniform
[none] .


--- Model how either one of the virtual machines, on either one of the physical servers
--- may crash, uniformly at random.
psdcontext CtxStrat := given state: CF:Configuration
                             rule: failVM
                               is: uniform
[none] .

psdsubst SubstStrat := given state: CF:Configuration
                             rule: failVM
                          context: CTX:Configuration
                               is: uniform
[none] .


--- Model the fact that the probability for two physical servers to take part in a VM migration
--- is large if the absolute difference between their failure ratios is large.
--- This strategy also (implicitly) associates a weight of 1, i.e., an uniform distribution
--- over the geographical regions and the different data center objects
--- inside which the VM migration takes place.
psdcontext CtxStrat :=
  given state: CF:Configuration
               < R:String : Region | ATTS:AttributeSet,
                 dataCenters : < DCO:Oid : DCenter |
                                 pservers : < PSO:Oid : PServer | ATTS':AttributeSet,
                                              load : M:Nat, vmFailed : VMFL:Nat >
                                            < PSO':Oid : PServer | ATTS'':AttributeSet,
                                              load : M':Nat, vmFailed : VMFL':Nat >
                                            PS:Configuration
                                 >
                                 DC:Configuration
               >
         rule: migrateVM
           is: (CF:Configuration
                < R:String : Region | ATTS:AttributeSet,
                  dataCenters : < DCO:Oid : DCenter | pservers : [] PS:Configuration >
                                DC:Configuration
                >) -> (abs(failureRatio(VMFL:Nat, M:Nat) - failureRatio(VMFL':Nat, M':Nat)))
[none] .
```

```
  --- Model that, given a pair of two, possibly unreliable servers,
  --- the probability for one to be the ``source'' server of the VM migration is larger
  --- if the server's failure ratio is large,
  --- whereas the probability for one to be the ``destination'' server is larger
  --- if the server's failure ratio is small.
  --- We also associate a higher probability to migrating a VM with large throughput T,
  --- and which belongs to a provider with high priority P.
  --- We model that priority is a more important factor, by squaring it in the weight expression.
  psdsubst SubstStrat :=
    given state: CF:Configuration
                 < O:Oid : SProvider | priority : P:Nat >
                 < R:String : Region | ATTS:AttributeSet,
                   dataCenters : < DCO:Oid : DCenter |
                                   pservers : <  XPSO:Oid : PServer | ATTS':AttributeSet,
                                                      load : XM:Nat,
                                                    vmLoad : XVMLD:Nat,
                                                  vmFailed : XVMFL:Nat,
                                                       vms : < XVMO:Oid : VMachine | ATTS'':AttributeSet,
                                                                      vmReq : XOIDSET:OidMSet,
                                                                    running : true,
                                                                      owner : O:Oid,
                                                                 throughput : T:Nat
                                                               >
                                                               XVM:Configuration
                                               >
                                               < XPSO':Oid : PServer | ATTS''':AttributeSet,
                                                      load : XM':Nat,
                                                   maxLoad : XN':Nat,
                                                    vmLoad : XVMLD':Nat,
                                                  vmFailed : XVMFL':Nat,
                                                       vms : XVM':Configuration
                                               >
                                               PS:Configuration
                                 >
                                 DC:Configuration
                 >
           rule: migrateVM
        context: CF:Configuration
                 < O:Oid : SProvider | priority : P:Nat >
                 < R:String : Region | ATTS:AttributeSet,
                   dataCenters : < DCO:Oid : DCenter | pservers : [] PS:Configuration >
                                 DC:Configuration >
             is: { PSO:Oid <- XPSO:Oid, M:Nat <- XM:Nat, VMLD:Nat <- XVMLD:Nat,
                   VMFL:Nat <- XVMFL:Nat, VMO:Oid <- XVMO:Oid, OIDSET:OidMSet <- XOIDSET:OidMSet,
                   VM:Configuration <- XVM:Configuration,
                   PSO':Oid <- XPSO':Oid, M':Nat <- XM':Nat,
                   VMLD':Nat <- XVMLD':Nat, N':Nat <- XN':Nat,
                   VMFL':Nat <- XVMFL':Nat, VM':Configuration <- XVM':Configuration }
                 -> ( P:Nat * P:Nat * (1 + T:Nat)
                      * ((1 + failureRatio(XVMFL:Nat, XM:Nat))
                         / (1 + failureRatio(XVMFL':Nat, XM':Nat))) )
  [none] .


  --- Put all of the above strategies together:
  psd Strat2b := < RuleStrat | CtxStrat | SubstStrat > .

endpsm)
```

D Python Script to Process Cloud Simulation Output

Below we give a Python script that processes the PSMaude output corresponding to a series of one-step probabilistic rewrites of the cloud computing model in Section 6, and generates a `.csv` file with the corresponding series of values for the cumulated computational effort.

```
# open files
f = open("<PSMaudeOutput>.txt","r")
g = open("<nameOfOutputFile>.csv","w")

line = f.readline()

while True:
  while True:
    line = f.readline()
    x = line.find("Result Configuration: eff(")
    if (x != -1):
      stats = line[x:]
      y1 = stats.find(")")
      g.write("= " + stats[26:y1] + "\n")
      while True:
        tmpline = f.readline()
        if tmpline == "\n":
          break
      break
  line = f.readline()
  if line == '' or line.find("Bye") != -1:
    break

# close files
f.close()
g.close()
```